

Covers C# 6 and 7



C# IN DEPTH

FOURTH EDITION

Jon Skeet

FOREWORD BY ERIC LIPPERT

MANNING

Praise for the Third Edition

“A must-have book that every .NET developer should read at least once.”

—Dror Helper, Software Architect, Better Place

“C# in Depth is the best source for learning C# language features.”

—Andy Kirsch, Software Architect, Venga

“Took my C# knowledge to the next level.”

—Dustin Laine, Owner, Code Harvest

“This book was quite an eye-opener to an interesting programming language that I have been unjustly ignoring until now.”

—Ivan Todorović, Senior Software Developer
AudatexGmbH, Switzerland

“Easily the best C# reference I’ve found.”

—Jon Parish, Software Engineer, Datasift

“Highly recommend this book to C# developers who want to take their knowledge to pro status.”

—D. Jay, Amazon reviewer

Praise for the Second Edition

“If you are looking to master C# then this book is a must-read.”

—Tyson S. Maxwell, Sr. Software Engineer, Raytheon

“We’re betting that this will be the best C# 4.0 book out there.”

—Nikander Bruggeman and Margriet Bruggeman
.NET consultants, Lois & Clark IT Services

“A useful and engaging insight into the evolution of C# 4.”

—Joe Albahari, Author of *LINQPad* and *C# 4.0 in a Nutshell*

“This book should be required reading for all professional C# developers.”

—Stuart Caborn, Senior Developer, BNP Paribas

“A highly focused, master-level resource on language updates across all major C# releases. This book is a must-have for the expert developer wanting to stay current with new features of the C# language.”

—Sean Reilly, Programmer/Analyst Point2 Technologies

“Why read the basics over and over again? Jon focuses on the chewy, new stuff!”

—Keith Hill, Software Architect, Agilent Technologies

“Everything you didn’t realize you needed to know about C#. ”

—Jared Parsons, Senior Software Development Engineer, Microsoft

Praise for the First Edition

“Simply put, C# in Depth is perhaps the best computer book I’ve read.”

—Craig Pelkie, Author, *System iNetwork*

“I have been developing in C# from the very beginning and this book had some nice surprises even for me. I was especially impressed with the excellent coverage of delegates, anonymous methods, covariance and contravariance. Even if you are a seasoned developer, C# in Depth will teach you something new about the C# language.... This book truly has depth that no other C# language book can touch.”

—Adam J. Wolf, Southeast Valley .NET User Group

“This book wraps up the author’s great knowledge of the inner workings of C# and hands it over to readers in a well-written, concise, usable book.”

—Jim Holmes, Author of *Windows Developer Power Tools*

“Every term is used appropriately and in the right context, every example is spot-on and contains the least amount of code that shows the full extent of the feature...this is a rare treat.”

—Franck Jeannin, Amazon UK reviewer

“If you have developed using C# for several years now, and would like to know the internals, this book is absolutely right for you.”

—Golo Roden
Author, Speaker, and Trainer for .NET and related technologies

“The best C# book I’ve ever read.”

—Chris Mullins, C# MVP

C# in Depth

FOURTH EDITION

JON SKEET
FOREWORD BY ERIC LIPPERT



MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

©2019 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

- ⊗ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.



Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor: Richard Wattenberger
Technical development editor: Dennis Sellinger
Review editor: Ivan Martinović
Production editor: Lori Weidert
Copyeditor: Sharon Wilkey
Technical proofreader: Eric Lippert
Typesetter and cover designer: Marija Tudor

ISBN 9781617294532

Printed in the United States of America

1 2 3 4 5 6 7 8 9 10 – SP – 24 23 22 21 20 19

This book is dedicated to equality, which is significantly harder to achieve in the real world than overriding Equals() and GetHashCode().

contents

<i>foreword</i>	xvii
<i>preface</i>	xix
<i>acknowledgments</i>	xx
<i>about this book</i>	xxii
<i>about the author</i>	xxvi
<i>about the cover illustration</i>	xxvii

PART 1 C# IN CONTEXT 1

I *Survival of the sharpest* 3

1.1 An evolving language 3

<i>A helpful type system at large and small scales</i>	4	▪ <i>Ever more concise code</i>	6	▪ <i>Simple data access with LINQ</i>	9
<i>Asynchrony</i>	10	▪ <i>Balancing efficiency and complexity</i>	11		
<i>Evolution at speed: Using minor versions</i>	12				

1.2 An evolving platform 13

1.3 An evolving community 14

1.4 An evolving book 15

<i>Mixed-level coverage</i>	16	▪ <i>Examples using Noda Time</i>	16
<i>Terminology choices</i>	17		

PART 2 C# 2–5 19**2 C# 2 21****2.1 Generics 22**

*Introduction by example: Collections before generics 22
Generics save the day 25 ▪ What can be generic? 29
Type inference for type arguments to methods 30 ▪ Type constraints 32 ▪ The default and typeof operators 34
Generic type initialization and state 37*

2.2 Nullable value types 38

Aim: Expressing an absence of information 39 ▪ CLR and framework support: The Nullable<T> struct 40 ▪ Language support 43

2.3 Simplified delegate creation 49

*Method group conversions 50 ▪ Anonymous methods 50
Delegate compatibility 52*

2.4 Iterators 53

*Introduction to iterators 54 ▪ Lazy execution 55 ▪ Evaluation of yield statements 56 ▪ The importance of being lazy 57
Evaluation of finally blocks 58 ▪ The importance of finally handling 61 ▪ Implementation sketch 62*

2.5 Minor features 66

*Partial types 67 ▪ Static classes 69 ▪ Separate getter/setter access for properties 69 ▪ Namespace aliases 70
Pragma directives 72 ▪ Fixed-size buffers 73
InternalsVisibleTo 73*

3 C# 3: LINQ and everything that comes with it 75**3.1 Automatically implemented properties 76****3.2 Implicit typing 77**

Typing terminology 77 ▪ Implicitly typed local variables (var) 78 ▪ Implicitly typed arrays 79

3.3 Object and collection initializers 81

*Introduction to object and collection initializers 81
Object initializers 83 ▪ Collection initializers 84
The benefits of single expressions for initialization 86*

3.4 Anonymous types 86

Syntax and basic behavior 86 ▪ The compiler-generated type 89 ▪ Limitations 90

3.5	Lambda expressions	91
	<i>Lambda expression syntax</i>	92 ▪ <i>Capturing variables</i> 94
	<i>Expression trees</i>	101
3.6	Extension methods	103
	<i>Declaring an extension method</i>	103 ▪ <i>Invoking an extension method</i> 104 ▪ <i>Chaining method calls</i> 106
3.7	Query expressions	107
	<i>Query expressions translate from C# to C#</i>	108 ▪ <i>Range variables and transparent identifiers</i> 108 ▪ <i>Deciding when to use which syntax for LINQ</i> 110
3.8	The end result: LINQ	111

4 C# 4: Improving interoperability 113

4.1	Dynamic typing	114
	<i>Introduction to dynamic typing</i>	114 ▪ <i>Dynamic behavior beyond reflection</i> 119 ▪ <i>A brief look behind the scenes</i> 124
	<i>Limitations and surprises in dynamic typing</i>	127 ▪ <i>Usage suggestions</i> 131
4.2	Optional parameters and named arguments	133
	<i>Parameters with default values and arguments with names</i>	134
	<i>Determining the meaning of a method call</i>	135 ▪ <i>Impact on versioning</i> 137
4.3	COM interoperability improvements	138
	<i>Linking primary interop assemblies</i>	139 ▪ <i>Optional parameters in COM</i> 140 ▪ <i>Named indexers</i> 142
4.4	Generic variance	143
	<i>Simple examples of variance in action</i>	143 ▪ <i>Syntax for variance in interface and delegate declarations</i> 144
	<i>Restrictions on using variance</i>	145 ▪ <i>Generic variance in practice</i> 147

5 Writing asynchronous code 150

5.1	Introducing asynchronous functions	152
	<i>First encounters of the asynchronous kind</i>	152 ▪ <i>Breaking down the first example</i> 154
5.2	Thinking about asynchrony	155
	<i>Fundamentals of asynchronous execution</i>	155 ▪ <i>Synchronization contexts</i> 157 ▪ <i>Modeling asynchronous methods</i> 158

5.3	Async method declarations	160
	<i>Return types from async methods</i>	161
	<i>Parameters in async methods</i>	162
5.4	Await expressions	162
	<i>The awaitable pattern</i>	163
	<i>Restrictions on await expressions</i>	165
5.5	Wrapping of return values	166
5.6	Asynchronous method flow	168
	<i>What is awaited and when?</i>	168
	<i>Evaluation of await expressions</i>	169
	<i>The use of awaitable pattern members</i>	173
	<i>Exception unwrapping</i>	174
	<i>Method completion</i>	176
5.7	Asynchronous anonymous functions	180
5.8	Custom task types in C# 7	182
	<i>The 99.9% case: ValueTask<TResult></i>	182
	<i>The 0.1% case: Building your own custom task type</i>	184
5.9	Async main methods in C# 7.1	186
5.10	Usage tips	187
	<i>Avoid context capture by using ConfigureAwait (where appropriate)</i>	187
	<i>Enable parallelism by starting multiple independent tasks</i>	189
	<i>Avoid mixing synchronous and asynchronous code</i>	190
	<i>Allow cancellation wherever possible</i>	190
	<i>Testing asynchrony</i>	191

6 Async implementation 193

6.1	Structure of the generated code	195
	<i>The stub method: Preparation and taking the first step</i>	198
	<i>Structure of the state machine</i>	199
	<i>The MoveNext() method (high level)</i>	202
	<i>The SetStateMachine method and the state machine boxing dance</i>	204
6.2	A simple MoveNext() implementation	205
	<i>A full concrete example</i>	205
	<i>MoveNext() method general structure</i>	207
	<i>Zooming into an await expression</i>	209
6.3	How control flow affects MoveNext()	210
	<i>Control flow between await expressions is simple</i>	211
	<i>Awaiting within a loop</i>	212
	<i>Awaiting within a try/finally block</i>	213
6.4	Execution contexts and flow	216
6.5	Custom task types revisited	218

7	C# 5 bonus features 220	
7.1	Capturing variables in foreach loops	220
7.2	Caller information attributes	222
	<i>Basic behavior</i>	222
	<i>Logging</i>	224
	<i>Simplifying INotifyPropertyChanged implementations</i>	224
	<i>Corner cases of caller information attributes</i>	226
	<i>Using caller information attributes with old versions of .NET</i>	232
PART 3	C# 6	233
8	Super-sleek properties and expression-bodied members 235	
8.1	A brief history of properties	236
8.2	Upgrades to automatically implemented properties	238
	<i>Read-only automatically implemented properties</i>	238
	<i>Initializing automatically implemented properties</i>	239
	<i>Automatically implemented properties in structs</i>	240
8.3	Expression-bodied members	242
	<i>Even simpler read-only computed properties</i>	242
	<i>Expression-bodied methods, indexers, and operators</i>	245
	<i>Restrictions on expression-bodied members in C# 6</i>	247
	<i>Guidelines for using expression-bodied members</i>	249
9	Stringy features 252	
9.1	A recap on string formatting in .NET	253
	<i>Simple string formatting</i>	253
	<i>Custom formatting with format strings</i>	253
	<i>Localization</i>	255
9.2	Introducing interpolated string literals	258
	<i>Simple interpolation</i>	258
	<i>Format strings in interpolated string literals</i>	259
	<i>Interpolated verbatim string literals</i>	259
	<i>Compiler handling of interpolated string literals (part 1)</i>	261
9.3	Localization using FormattableString	261
	<i>Compiler handling of interpolated string literals (part 2)</i>	262
	<i>Formatting a FormattableString in a specific culture</i>	263
	<i>Other uses for FormattableString</i>	265
	<i>Using FormattableString with older versions of .NET</i>	268
9.4	Uses, guidelines, and limitations	270
	<i>Developers and machines, but maybe not end users</i>	270
	<i>Hard limitations of interpolated string literals</i>	272
	<i>When you can but really shouldn't</i>	273

9.5	Accessing identifiers with nameof	275
	<i>First examples of nameof</i>	275
	<i>Common uses of nameof</i>	277
	<i>Tricks and traps when using nameof</i>	280
10	A smörgåsbord of features for concise code	284
10.1	Using static directives	284
	<i>Importing static members</i>	285
	<i>Extension methods and using static</i>	288
10.2	Object and collection initializer enhancements	290
	<i>Indexers in object initializers</i>	291
	<i>Using extension methods in collection initializers</i>	294
	<i>Test code vs. production code</i>	298
10.3	The null conditional operator	299
	<i>Simple and safe property dereferencing</i>	299
	<i>The null conditional operator in more detail</i>	300
	<i>Handling Boolean comparisons</i>	301
	<i>Indexers and the null conditional operator</i>	302
	<i>Working effectively with the null conditional operator</i>	303
	<i>Limitations of the null conditional operator</i>	305
10.4	Exception filters	305
	<i>Syntax and semantics of exception filters</i>	306
	<i>Retrying operations</i>	311
	<i>Logging as a side effect</i>	312
	<i>Individual, case-specific exception filters</i>	313
	<i>Why not just throw?</i>	314
PART 4	C# 7 AND BEYOND	317
11	Composition using tuples	319
11.1	Introduction to tuples	320
11.2	Tuple literals and tuple types	321
	<i>Syntax</i>	321
	<i>Inferred element names for tuple literals (C# 7.1)</i>	323
	<i>Tuples as bags of variables</i>	324
11.3	Tuple types and conversions	329
	<i>Types of tuple literals</i>	329
	<i>Conversions from tuple literals to tuple types</i>	330
	<i>Conversions between tuple types</i>	334
	<i>Uses of conversions</i>	336
	<i>Element name checking in inheritance</i>	336
	<i>Equality and inequality operators (C# 7.3)</i>	337
11.4	Tuples in the CLR	338
	<i>Introducing System.ValueTuple<...></i>	338
	<i>Element name handling</i>	339
	<i> Tuple conversion implementations</i>	341
	<i>String representations of tuples</i>	341
	<i>Regular equality and ordering comparisons</i>	342
	<i>Structural equality and ordering</i>	

comparisons 343 ▪ *Wombles and large tuples* 345 ▪ *The nongeneric ValueTuple struct* 346 ▪ *Extension methods* 346

11.5 Alternatives to tuples 346

System.Tuple<...> 347 ▪ *Anonymous types* 347
Named types 348

11.6 Uses and recommendations 348

Nonpublic APIs and easily changed code 348 ▪ *Local variables* 349 ▪ *Fields* 350 ▪ *Tuples and dynamic don't play together nicely* 351

12 Deconstruction and pattern matching 353

12.1 Deconstruction of tuples 354

Deconstruction to new variables 355 ▪ *Deconstruction assignments to existing variables and properties* 357
Details of tuple literal deconstruction 361

12.2 Deconstruction of nontuple types 361

Instance deconstruction methods 362 ▪ *Extension deconstruction methods and overloading* 363 ▪ *Compiler handling of Deconstruct calls* 364

12.3 Introduction to pattern matching 365

12.4 Patterns available in C# 7.0 367

Constant patterns 367 ▪ *Type patterns* 368 ▪ *The var pattern* 371

12.5 Using patterns with the is operator 372

12.6 Using patterns with switch statements 374

Guard clauses 375 ▪ *Pattern variable scope for case labels* 376 ▪ *Evaluation order of pattern-based switch statements* 377

12.7 Thoughts on usage 379

Spotting deconstruction opportunities 379 ▪ *Spotting pattern matching opportunities* 380

13 Improving efficiency with more pass by reference 381

13.1 Recap: What do you know about ref? 382

13.2 Ref locals and ref returns 385

Ref locals 385 ▪ *Ref returns* 390 ▪ *The conditional ?: operator and ref values (C# 7.2)* 392 ▪ *Ref readonly (C# 7.2)* 393

13.3 in parameters (C# 7.2) 395

Compatibility considerations 396 ▪ *The surprising mutability of in parameters: External changes* 397 ▪ *Overloading with in parameters* 398 ▪ *Guidance for in parameters* 399

- 13.4 Declaring structs as readonly (C# 7.2) 401
Background: Implicit copying with read-only variables 401
The readonly modifier for structs 403 ▪ *XML serialization is implicitly read-write* 404
- 13.5 Extension methods with ref or in parameters (C# 7.2) 405
Using ref/in parameters in extension methods to avoid copying 405
Restrictions on ref and in extension methods 407
- 13.6 Ref-like structs (C# 7.2) 408
Rules for ref-like structs 409 ▪ *Span<T> and stackalloc* 410
IL representation of ref-like structs 414

14 Concise code in C# 7 415

- 14.1 Local methods 415
Variable access within local methods 417 ▪ *Local method implementations* 420 ▪ *Usage guidelines* 425
- 14.2 Out variables 427
Inline variable declarations for out parameters 427 ▪ *Restrictions listed in C# 7.3 for out variables and pattern variables* 428
- 14.3 Improvements to numeric literals 429
Binary integer literals 429 ▪ *Underscore separators* 430
- 14.4 Throw expressions 431
- 14.5 Default literals (C# 7.1) 432
- 14.6 Nontrailing named arguments (C# 7.2) 433
- 14.7 Private protected access (C# 7.2) 435
- 14.8 Minor improvements in C# 7.3 435
Generic type constraints 435 ▪ *Overload resolution improvements* 436 ▪ *Attributes for fields backing automatically implemented properties* 437

15 C# 8 and beyond 439

- 15.1 Nullable reference types 440
What problem do nullable reference types solve? 440 ▪ *Changing the meaning when using reference types* 441 ▪ *Enter nullable reference types* 442 ▪ *Nullable reference types at compile time and*

	<i>execution time</i>	443	▪ <i>The damn it or bang operator</i>	445
	<i>Experiences of nullable reference type migration</i>	447		
	<i>Future improvements</i>	449		
15.2	Switch expressions	453		
15.3	Recursive pattern matching	455		
	<i>Matching properties in patterns</i>	455	▪ <i>Deconstruction patterns</i>	456
			▪ <i>Omitting types from patterns</i>	457
15.4	Indexes and ranges	458		
	<i>Index and Range types and literals</i>	458	▪ <i>Applying indexes and ranges</i>	459
15.5	More async integration	461		
	<i>Asynchronous resource disposal with using await</i>	461		
	<i>Asynchronous iteration with foreach await</i>	462	▪ <i>Asynchronous iterators</i>	465
15.6	Features not yet in preview	466		
	<i>Default interface methods</i>	466	▪ <i>Record types</i>	468
	<i>Even more features in brief</i>	469		
15.7	Getting involved	470		
appendix	<i>Language features by version</i>	473		
	<i>index</i>	479		

foreword

Ten years is a long stretch of time for a human, and it's an absolute eternity for a technical book aimed at professional programmers. It was with some astonishment, then, that I realized 10 years have passed since Microsoft shipped C# 3.0 with Visual Studio 2008 and since I read the drafts of the first edition of this book. It has also been 10 years since Jon joined Stack Overflow and quickly became the user with the highest reputation.

C# was already a large, complex language in 2008, and the design and implementation teams haven't been idle for the last decade. I'm thrilled with how C# has been innovative in meeting the needs of many different developer constituencies, from video games to websites to low-level, highly robust system components. C# takes the best from academic research and marries it to practical techniques for solving real problems. It's not dogmatic; the C# designers don't ask "What's the most object-oriented way to design this feature?" or "What's the most functional way to design this feature?" but rather "What's the most pragmatic, safe, and effective way to design this feature?" Jon gets all of that. He doesn't just explain how the language works; he explains how the whole thing holds together as a unified design and also points out when it doesn't.

I said in my foreword to the first edition that Jon is enthusiastic, knowledgeable, talented, curious, analytical, and a great teacher, and all of that is still true. Let me add to that list by noting his perseverance and dedication. Writing a book is a huge job, particularly when you do it in your spare time. Going back and revising that book to keep it fresh and current is just as much work, and this is the third time Jon has done that with this book. A lesser author would be content to tweak it here and there or add

a chapter about new materials; this is more like a large-scale refactoring. The results speak for themselves.

More than ever, I can't wait to find out what great things the next generation of programmers will do with C# as it continues to evolve and grow. I hope you enjoy this book as much as I have over the years, and thanks for choosing to compose your programs in C#.

ERIC LIPPERT
SOFTWARE ENGINEER
FACEBOOK

****preface****

Welcome to the fourth edition of *C# in Depth*. When I wrote the first edition, I had little idea I'd be writing a fourth edition of the same title 10 years later. Now, it wouldn't surprise me to find myself writing another edition in 10 years. Since the first edition, the designers of the C# language have repeatedly proved that they're dedicated to evolving the language for as long as the industry is interested in it.

This is important, because the industry has changed a lot in the last 10 years. As a reminder, both the mobile ecosystem (as we know it today) and cloud computing were still in their infancy in 2008. Amazon EC2 was launched in 2006, and Google AppEngine was launched in 2008. Xamarin was launched by the Mono team in 2011. Docker didn't show up until 2013.

For many .NET developers, the really big change in our part of the computing world over the last few years has been .NET Core. It's a cross-platform, open source version of the framework that is explicitly designed for compatibility with other frameworks (via .NET Standard). Its existence is enough to raise eyebrows; that it is Microsoft's primary area of investment in .NET is even more surprising.

Through all of this, C# is still the primary language when targeting anything like .NET, whether that's .NET, .NET Core, Xamarin, or Unity. F# is a healthy and friendly competitor, but it doesn't have the industry mindshare of C#.

I've personally been developing in C# since around 2002, either professionally or as an enthusiastic amateur. As the years have gone by, I've been sucked ever deeper into the details of the language. I enjoy those details for their own sake but, more importantly, for the sake of ever-increasing productivity when writing code in C#. I hope that some of that enjoyment has seeped into this book and will encourage you further in your travels with C#.

acknowledgments

It takes a lot of work and energy to create a book. Some of that is obvious; after all, pages don't just write themselves. That's just the tip of the iceberg, though. If you received the first version of the content I wrote with no editing, no review, no professional typesetting, and so on, I suspect you'd be pretty disappointed.

As with previous editions, it's been a pleasure working with the team at Manning. Richard Wattenberger has provided guidance and suggestions with just the right combination of insistence and understanding, thereby shaping the content through multiple iterations. (In particular, working out the best approach to use for C# 2–4 proved surprisingly challenging.) I would also like to thank Mike Stephens and Marjan Bace for supporting this edition from the start.

Beyond the structure of the book, the review process is crucial to keeping the content accurate and clear. Ivan Martinovic organized the peer reviewing process and obtained great feedback from Ajay Bhosale, Andrei Rînea, Andy Kirsch, Brian Rasmussen, Chris Heneghan, Christos Paisios, Dmytro Lypai, Ernesto Cardenas, Gary Hubbard, Jassel Holguin Calderon, Jeremy Lange, John Meyer, Jose Luis Perez Vila, Karl Metivier, Meredith Godar, Michal Paszkiewicz, Mikkel Arentoft, Nelson Ferrari, Prajwal Khanal, Rami Abdelwahed, and Willem van Ketwicha. I'm indebted to Dennis Sellinger for his technical editing and to Eric Lippert for technical proofreading. I want to highlight Eric's contributions to every edition of this book, which have always gone well beyond technical corrections. His insight, experience, and humor have been significant and unexpected bonuses throughout the whole process.

Content is one thing; good-looking content is another. Lori Weidert managed the complex production process with dedication and understanding. Sharon Wilkey performed copyediting with skill and the utmost patience. The typesetting and cover

design were done by Marija Tudor, and I can't express what a joy it is to see the first typeset pages; it's much like the first (successful) dress rehearsal of a play you've been working on for months.

Beyond the people who've contributed directly to the book, I naturally need to thank my family for continuing to put up with me over the last few years. I love my family. They rock, and I'm grateful.

Finally, none of this would matter if no one wanted to read the book. Thank you for your interest, and I hope your investment of time into this book pays off.

about this book

Who should read this book

This book is about the *language* of C#. That often means going into some details of the runtime responsible for executing your code and the libraries that support your application, but the focus is firmly on the language itself.

The goal of the book is to make you as comfortable as possible with C# so you never need to feel you're fighting against it. I want to help you feel you are *fluent* in C#, with the associated connotations of working in a fluid and flowing way. Think of C# as a river in which you're paddling a kayak. The better you know the river, the faster you'll be able to travel with its flow. Occasionally, you'll want to paddle upstream for some reason; even then, knowing how the river moves will make it easier to reach your target without capsizing.

If you're an existing C# programmer who wants to know more about the language, this book is for you! You don't need to be an expert to read this book, but I assume you know the basics of C# 1. I explain all the terminology I use that was introduced after C# 1 and some older terms that are often misunderstood (such as parameters and arguments), but I assume you know what a class is, what an object is, and so on.

If you are an expert already, you may still find the book useful because it provides different ways of thinking about concepts that are already familiar to you. You may also discover areas of the language you were unaware of; I know that's been my experience in writing the book.

If you're completely new to C#, this book may not be useful to you *yet*. There are a lot of introductory books and online tutorials on C#. Once you have a grip on the basics, I hope you'll return here and dive deeper.

How this book is organized: A roadmap

This book comprises 15 chapters divided into 4 parts. Part 1 provides a brief history of the language.

- Chapter 1 gives an overview of how C# has changed over the years and how it is still changing. It puts C# into a broader context of platforms and communities and gives a little more detail about how I present material in the rest of the book.

Part 2 describes C# versions 2 through 5. This is effectively a rewritten and condensed form of the third edition of this book.

- Chapter 2 demonstrates the wide variety of features introduced in C# 2, including generics, nullable value types, anonymous methods, and iterators.
- Chapter 3 explains how the features of C# 3 come together to form LINQ. The most prominent features in this chapter are lambda expressions, anonymous types, object initializers, and query expressions.
- Chapter 4 describes the features of C# 4. The largest change within C# 4 was the introduction of dynamic typing, but there are other changes around optional parameters, named arguments, generic variance, and reducing friction when working with COM.
- Chapter 5 begins the coverage of C# 5's primary feature: `async/await`. This chapter describes how you'll use `async/await` but has relatively little detail about how it works behind the scenes. Enhancements to asynchrony introduced in later versions of C# are described here as well, including custom task types and `async` main methods.
- Chapter 6 completes the `async/await` coverage by going deep into the details of how the compiler handles asynchronous methods by creating state machines.
- Chapter 7 is a short discussion of the few features introduced in C# 5 besides `async/await`. After the all the details provided in chapter 6, you can consider it a palette cleanser before moving on to the next part of the book.

Part 3 describes C# 6 in detail.

- Chapter 8 shows expression-bodied members, which allow you to remove some of the tedious syntax when declaring very simple properties and methods. Improvements to automatically implemented properties are described here, too. It's all about streamlining your source code.
- Chapter 9 describes the string-related features of C# 6: interpolated string literals and the `nameof` operator. Although both features are just new ways of producing strings, they are among the most handy aspects of C# 6.
- Chapter 10 introduces the remaining features of C# 6. These have no particularly common theme other than helping you write concise source code. Of the

features introduced here, the null conditional operator is probably the most useful; it's a clean way of short-circuiting expressions that might involve null values, thereby avoiding the dreaded `NullReferenceException`.

Part 4 addresses C# 7 (all the way up to C# 7.3) and completes the book by peering a short distance into the future.

- Chapter 11 demonstrates the integration of tuples into the language and describes the `ValueTuple` family of types that is used for the implementation.
- Chapter 12 introduces deconstruction and pattern matching. These are both concise ways of looking at an existing value in a different way. In particular, pattern matching in switch statements can simplify how you handle different types of values in situations where inheritance doesn't quite fit.
- Chapter 13 focuses on pass by reference and related features. Although `ref` parameters have been present in C# since the very first version, C# 7 introduces a raft of new features such as `ref` returns and `ref` locals. These are primarily aimed at improving efficiency by reducing copying.
- Chapter 14 completes the C# 7 coverage with another set of small features that all contribute to streamlining your code. Of these, my personal favorites are local methods, `out` variables, and the default literal, but there are other little gems to discover, too.
- Chapter 15 looks at the future of C#. Working with the C# 8 preview available at the time of this writing, I delve into nullable reference types, switch expressions, and pattern matching enhancements as well as ranges and further integration of asynchrony into core language features. This entire chapter is speculative, but I hope it will spark your curiosity.

Finally, the appendix provides a handy reference for which features were introduced in which version of C# and whether they have runtime or framework requirements that restrict the contexts in which you can use them.

My expectation is that this book will be read in a linear fashion (at least the first time). Later chapters build on earlier ones, and you may have a hard time if you try to read them out of order. After you've read the book once, however, it makes perfect sense to use it as a reference. You might go back to a topic when you need a reminder of some syntax or if you find yourself caring more about a specific detail than you did on your first reading.

About the code

This book contains many examples of source code in numbered listings and in line with normal text. In both cases, source code is formatted in a fixed-width font like this to separate it from ordinary text. Sometimes it appears **in bold** to highlight code that has changed from previous steps in the chapter, such as when a new feature adds to an existing line of code.

In many cases, the original source code has been reformatted; I've added line breaks and reworked indentation to accommodate the available page space in the book. In rare cases, listings include line-continuation markers (➡). In addition, comments in the source code have often been removed from the listings when the code is described in the text. Code annotations accompany many of the listings and highlight important concepts.

Source code for the examples in this book is available for download from the publisher's website at www.manning.com/books/c-sharp-in-depth-fourth-edition. You'll need the .NET Core SDK (version 2.1.300 or higher) installed to build the examples. A few examples require the Windows desktop .NET framework (where Windows Forms or COM is involved), but most are portable via .NET Core. Although I used Visual Studio 2017 (Community Edition) to develop the examples, they should be fine under Visual Studio Code as well.

Book forum

Purchase of *C# in Depth*, Fourth Edition, includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the author and from other users. To access the forum, go to <https://forums.manning.com/forums/c-sharp-in-depth-fourth-edition>. You can also learn more about Manning's forums and the rules of conduct at <https://forums.manning.com/forums/about>.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the author some challenging questions lest his interest stray! The forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

Other online resources

There are many, many resources for C# online. The ones I find most useful are listed below, but you'll find a lot more by searching, too.

- Microsoft .NET documentation: <https://docs.microsoft.com/dotnet>
- The .NET API documentation: <https://docs.microsoft.com/dotnet/api>
- The C# language design repository: <https://github.com/dotnet/csharplang>
- The Roslyn repository: <https://github.com/dotnet/roslyn>
- The C# ECMA standard:
www.ecma-international.org/publications/standards/Ecma-334.htm
- Stack Overflow: <https://stackoverflow.com>

about the author

My name is Jon Skeet. I'm a staff software engineer at Google, and I work from the London office. Currently, my role is to provide .NET client libraries for Google Cloud Platform, which neatly combines my enthusiasm for working at Google with my love of C#. I'm the convener of the ECMA technical group responsible for standardizing C#, and I represent Google within the .NET Foundation.

I'm probably best known for my contributions on Stack Overflow, which is a question-and-answer site for developers. I also enjoy speaking at conferences and user groups and blogging. The common factor here is interacting with other developers; it's the way I learn best.

Slightly more unusually, I'm a date and time hobbyist. This is mostly expressed through my work on Noda Time, which is the date and time library for .NET that you'll see used in several examples in this book. Even without the hands-on coding aspect, time is a fascinating topic with an abundance of trivia. Find me at a conference and I'll bore you for as long as you like about time zones and calendar systems.

My editors would like you to know most of these things to prove that I'm qualified to write this book, but please don't mistake them for a claim of infallibility. Humility is a vital part of being an effective software engineer, and I screw up just like everyone else does. Compilers don't tend to view appeals to authority in a favorable light.

In the book, I've tried to make it clear where I'm expressing what I believe to be objective facts about the C# language and where I'm expressing my opinion. Due to diligent technical reviewers, I hope there are relatively few mistakes on the objective side, but experience from previous editions suggests that some errors will have crept through. When it comes to opinions, mine may be wildly different from yours, and that's fine. Take what you find useful, and feel free to ignore the rest.

about the cover illustration

The caption for the illustration on the cover of *C# in Depth*, Fourth Edition, is “Musician.” The illustration is taken from a collection of costumes of the Ottoman Empire published on January 1, 1802, by William Miller of Old Bond Street, London. The title page is missing from the collection, and we have been unable to track it down to date. The book’s table of contents identifies the figures in both English and French, and each illustration bears the names of two artists who worked on it, both of whom would no doubt be surprised to find their art gracing the front cover of a computer programming book...two hundred years later.

The collection was purchased by a Manning editor at an antiquarian flea market in the “Garage” on West 26th Street in Manhattan. The seller was an American based in Ankara, Turkey, and the transaction took place just as he was packing up his stand for the day. The Manning editor didn’t have on his person the substantial amount of cash that was required for the purchase, and a credit card and check were both politely turned down. With the seller flying back to Ankara that evening, the situation was getting hopeless. What was the solution? It turned out to be nothing more than an old-fashioned verbal agreement sealed with a handshake. The seller simply proposed that the money be transferred to him by wire, and the editor walked out with the bank information on a piece of paper and the portfolio of images under his arm. Needless to say, he transferred the funds the next day, and he remains grateful and impressed by this unknown person’s trust. It recalls something that might have happened a long time ago.

We at Manning celebrate the inventiveness, the initiative, and, yes, the fun of the computer business with book covers based on the rich diversity of regional life of two centuries ago brought back to life by the pictures from this collection.

Part 1

C# in context

When I was studying computer science at university, a fellow student corrected the lecturer about a detail he'd written on the blackboard. The lecturer looked mildly exasperated and answered, "Yes, I know. I was simplifying. I'm obscuring the truth here to demonstrate a bigger truth." Although I hope I'm not obscuring much in part 1, it's definitely about the bigger truth.

Most of this book looks at C# close up, occasionally putting it under a microscope to see the finest details. Before we start doing that, chapter 1 pulls back the lens to see the broader sweep of the history of C# and how C# fits into the wider context of computing.

You'll see some code as an appetizer before I serve the main course of the rest of the book, but the details don't matter at this stage. This part is more about the ideas and themes of C#'s development to get you in the best frame of mind to appreciate how those ideas are implemented.

Let's go!

1

Survival of the sharpest

This chapter covers

- How C#'s rapid evolution has made developers more productive
- Selecting minor versions of C# to use the latest features
- Being able to run C# in more environments
- Benefiting from an open and engaged community
- The book's focus on old and new C# versions

Choosing the most interesting aspects of C# to introduce here was difficult. Some are fascinating but are rarely used. Others are incredibly important but are now commonplace to C# developers. Features such as `async/await` are great in many ways but are hard to describe briefly. Without further ado, let's look at how far C# has come over time.

1.1 An evolving language

In previous editions of this book, I provided a single example that showed the evolution of the language over the versions covered by that edition. That's no longer feasible in a way that would be interesting to read. Although a large application

may use almost all of the new features, any single piece of code that's suitable for the printed page would use only a subset of them.

Instead, in this section I choose what I consider to be the most important themes of C# evolution and give brief examples of improvements. This is far from an exhaustive list of features. It's also not intended to teach you the features; instead, it's a reminder of how far features you already know about have improved the language and a tease for features you may not have seen yet.

If you think some of these features imitate other languages you're familiar with, you're almost certainly right. The C# team does not hesitate to take great ideas from other languages and reshape them to feel at home within C#. This is a great thing! F# is particularly worth mentioning as a source of inspiration for many C# features.

NOTE It's possible that F#'s greatest impact isn't what it enables for F# developers but its influence on C#. This isn't to underplay the value of F# as a language in its own right or to suggest that it shouldn't be used directly. But currently, the C# community is significantly larger than the F# community, and the C# community owes a debt of gratitude to F# for inspiring the C# team.

Let's start with one of the most important aspects of C#: its type system.

1.1.1 A helpful type system at large and small scales

C# has been a statically typed language from the start: your code specifies the types of variables, parameters, values returned from methods, and so on. The more precisely you can specify the shape of the data your code accepts and returns, the more the compiler can help you avoid mistakes.

That's particularly true as the application you're building grows. If you can see all the code for your whole program on one screen (or at least hold it all in your head at one time), a statically typed language doesn't have much benefit. As the scale increases, it becomes increasingly important that your code concisely and effectively communicates what it does. You can do that through documentation, but static typing lets you communicate in a machine-readable way.

As C# has evolved, its type system has allowed more fine-grained descriptions. The most obvious example of this is *generics*. In C# 1, you might have had code like this:

```
public class Bookshelf
{
    public IEnumerable Books { get { ... } }
}
```

What type is each item in the `Books` sequence? The type system doesn't tell you. With generics in C# 2, you can communicate more effectively:

```
public class Bookshelf
{
    public IEnumerable<Book> Books { get { ... } }
}
```

C# 2 also brought *nullable value types*, thereby allowing the absence of information to be expressed effectively without resorting to magic values such as -1 for a collection index or `DateTime.MinValue` for a date.

C# 7 gave us the ability to tell the compiler that a user-defined struct should be immutable using `readonly struct` declarations. The primary goal for this feature may have been to improve the efficiency of the code generated by the compiler, but it has additional benefits for communicating intent.

The plans for C# 8 include *nullable reference types*, which will allow even more communication. Up to this point, nothing in the language lets you express whether a reference (either as a return value, a parameter, or just a local variable) might be null. This leads to error-prone code if you're not careful and boilerplate validation code if you are careful, neither of which is ideal. C# 8 will expect that anything not explicitly nullable is intended not to be nullable. For example, consider a method declaration like this:

```
string Method(string x, string? y)
```

The parameter types indicate that the argument corresponding to `x` shouldn't be null but that the argument corresponding to `y` may be null. The return type indicates that the method won't return null.

Other changes to the type system in C# are aimed at a smaller scale and focus on how one method might be implemented rather than how different components in a large system relate to each other. C# 3 introduced *anonymous types* and *implicitly typed local variables* (`var`). These help address the downside of some statically typed languages: verbosity. If you need a particular data shape within a single method but nowhere else, creating a whole extra type just for the sake of that method is overkill. Anonymous types allow that data shape to be expressed concisely without losing the benefits of static typing:

```
var book = new { Title = "Lost in the Snow", Author = "Holly Webb" };
string title = book.Title;
string author = book.Author;
```

 Name and type are still checked by the compiler

Anonymous types are primarily used within LINQ queries, but the principle of creating a type just for a single method doesn't depend on LINQ.

Similarly, it seems redundant to explicitly specify the type of a variable that is initialized in the same statement by calling the constructor of that type. I know which of the following declarations I find cleaner:

```
Dictionary<string, string> map1 = new Dictionary<string, string>();
```

```
var map2 = new Dictionary<string, string>();
```

 Explicit typing
Implicit typing

Although implicit typing is necessary when working with anonymous types, I've found it increasingly useful when working with regular types, too. It's important to distinguish

between *implicit* typing and *dynamic* typing. The preceding `map2` variable is still statically typed, but you didn't have to write the type explicitly.

Anonymous types help only within a single block of code; for example, you can't use them as method parameters or return types. C# 7 introduced *tuples*: value types that effectively act to collect variables together. The framework support for these tuples is relatively simple, but additional language support allows the elements of tuples to be named. For example, instead of the preceding anonymous type, you could use the following:

```
var book = (title: "Lost in the Snow", author: "Holly Webb");
Console.WriteLine(book.title);
```

Tuples can replace anonymous types in some cases but certainly not all. One of their benefits is that they *can* be used as method parameters and return types. At the moment, I advise that these be kept within the internal API of a program rather than exposed publicly, because tuples represent a simple composition of values rather than encapsulating them. That's why I still regard them as contributing to simpler code at the implementation level rather than improving overall program design.

I should mention a feature that *might* come in C# 8: *record types*. I think of these as named anonymous types to some extent, at least in their simplest form. They'd provide the benefits of anonymous types in terms of removing boilerplate code but then allow those types to gain extra behavior just as regular classes do. Watch this space!

1.1.2 Ever more concise code

One of the recurring themes within new features of C# has been the ability to let you express your ideas in ways that are increasingly concise. The type system is part of this, as you've seen with anonymous types, but many other features also contribute to this. There are lots of words you might hear for this, especially in terms of what can be removed with the new features in place. C#'s features allow you to reduce *ceremony*, remove *boilerplate* code, and avoid *cruft*. These are just different ways of talking about the same effect. It's not that any of the now-redundant code was wrong; it was just distracting and unnecessary. Let's look at a few ways that C# has evolved in this respect.

CONSTRUCTION AND INITIALIZATION

First, we'll consider how you create and initialize objects. Delegates have probably evolved the most and in multiple stages. In C# 1, you had to write a separate method for the delegate to refer to and then create the delegate itself in a long-winded way. For example, here's what you'd write to subscribe a new event handler to a button's `Click` event in C# 1:

```
button.Click += new EventHandler(HandleButtonClick); ← C# 1
```

C# 2 introduced *method group conversions* and *anonymous methods*. If you wanted to keep the `HandleButtonClick` method, method group conversions would allow you to change the preceding code to the following:

```
button.Click += HandleButtonClick; ← C# 2
```

If your click handler is simple, you might not want to bother with a separate method at all and instead use an anonymous method:

```
button.Click += delegate { MessageBox.Show("Clicked!"); }; ← C# 2
```

Anonymous methods have the additional benefit of acting as *closures*: they can use local variables in the context within which they're created. They're not used often in modern C# code, however, because C# 3 provided us with *lambda expressions*, which have almost all the benefits of anonymous methods but shorter syntax:

```
button.Click += (sender, args) => MessageBox.Show("Clicked!"); ← C# 3
```

NOTE In this case, the lambda expression is longer than the anonymous method because the anonymous method uses the one feature that lambda expressions don't have: the ability to ignore parameters by not providing a parameter list.

I used event handlers as an example for delegates because that was their main use in C# 1. In later versions of C#, delegates are used in more varied situations, particularly in LINQ.

LINQ also brought other benefits for initialization in the form of *object initializers* and *collection initializers*. These allow you to specify a set of properties to set on a new object or items to add to a new collection within a single expression. It's simpler to show than describe, and I'll borrow an example from chapter 3. Consider code that you might previously have written like this:

```
var customer = new Customer();
customer.Name = "Jon";
customer.Address = "UK";
var item1 = new OrderItem();
item1.ItemId = "abcd123";
item1.Quantity = 1;
var item2 = new OrderItem();
item2.ItemId = "fghi456";
item2.Quantity = 2;
var order = new Order();
order.OrderId = "xyz";
order.Customer = customer;
order.Items.Add(item1);
order.Items.Add(item2);
```

The object and collection initializers introduced in C# 3 make this so much clearer:

```
var order = new Order
{
    OrderId = "xyz",
    Customer = new Customer { Name = "Jon", Address = "UK" },
    Items =
    {
        new OrderItem { ItemId = "abcd123", Quantity = 1 },
        new OrderItem { ItemId = "fghi456", Quantity = 2 }
    }
};
```

I don't suggest reading either of these examples in detail; what's important is the simplicity of the second form over the first.

METHOD AND PROPERTY DECLARATIONS

One of the most obvious examples of simplification is through *automatically implemented properties*. These were first introduced in C# 3 but have been further improved in later versions. Consider a property that would've been implemented in C# 1 like this:

```
private string name;
public string Name
{
    get { return name; }
    set { name = value; }
}
```

Automatically implemented properties allow this to be written as a single line:

```
public string Name { get; set; }
```

Additionally, C# 6 introduced *expression-bodied members* that remove more ceremony. Suppose you're writing a class that wraps an existing collection of strings, and you want to effectively delegate the Count and GetEnumerator() members of your class to that collection. Prior to C# 6, you would've had to write something like this:

```
public int Count { get { return list.Count; } }

public IEnumerator<string> GetEnumerator()
{
    return list.GetEnumerator();
}
```

This is a strong example of ceremony: a lot of syntax that the language used to require with little benefit. In C# 6, this is significantly cleaner. The => syntax (already used by lambda expressions) is used to indicate an expression-bodied member:

```
public int Count => list.Count;

public IEnumerator<string> GetEnumerator() => list.GetEnumerator();
```

Although the value of using expression-bodied members is a personal and subjective matter, I've been surprised by just how much difference they've made to the readability of my code. I love them! Another feature I hadn't expected to use as much as I now do is string interpolation, which is one of the string-related improvements in C#.

STRING HANDLING

String handling in C# has had three significant improvements:

- C# 5 introduced *caller information attributes*, including the ability for the compiler to automatically populate method and filenames as parameter values. This is great for diagnostic purposes, whether in permanent logging or more temporary testing.

- C# 6 introduced the `nameof` operator, which allows names of variables, types, methods, and other members to be represented in a refactoring-friendly form.
- C# 6 also introduced *interpolated string literals*. This isn't a new concept, but it makes constructing a string with dynamic values much simpler.

For the sake of brevity, I'll demonstrate just the last point. It's reasonably common to want to construct a string with variables, properties, the result of method calls, and so forth. This might be for logging purposes, user-oriented error messages (if localization isn't required), exception messages, and so forth.

Here's an example from my Noda Time project. Users can try to find a calendar system by its ID, and the code throws a `KeyNotFoundException` if that ID doesn't exist. Prior to C# 6, the code might have looked like this:

```
throw new KeyNotFoundException(
    "No calendar system for ID " + id + " exists");
```

Using explicit string formatting, it looks like this:

```
throw new KeyNotFoundException(
    string.Format("No calendar system for ID {0} exists", id));
```

NOTE See section 1.4.2 for information about Noda Time. You don't need to know about it to understand this example.

In C# 6, the code becomes just a little simpler with an interpolated string literal to include the value of `id` in the string directly:

```
throw new KeyNotFoundException($"No calendar system for ID {id} exists");
```

This doesn't look like a big deal, but I'd hate to have to work without string interpolation now.

These are just the most prominent features that help improve the signal-to-noise ratio of your code. I could've shown using `static` directives and the null conditional operator in C# 6 as well as pattern matching, deconstruction, and `out` variables in C# 7. Rather than expand this chapter to mention every feature in every version, let's move on to a feature that's more revolutionary than evolutionary: LINQ.

1.1.3 Simple data access with LINQ

If you ask C# developers what they love about C#, they'll likely mention LINQ. You've already seen some of the features that build up to LINQ, but the most radical is query expressions. Consider this code:

```
var offers =
    from product in db.Products
    where product.SalePrice <= product.Price / 2
    orderby product.SalePrice
    select new {
        product.Id, product.Description,
        product.SalePrice, product.Price
    };
```

That doesn't look anything like old-school C#. Imagine traveling back to 2007 to show that code to a developer using C# 2 and then explaining that this has compile-time checking and IntelliSense support and that it results in an efficient database query. Oh, and that you can use the same syntax for regular collections as well.

Support for querying out-of-process data is provided via *expression trees*. These represent code as data, and a LINQ provider can analyze the code to convert it into SQL or other query languages. Although this is extremely cool, I rarely use it myself, because I don't work with SQL databases often. I do work with in-memory collections, though, and I use LINQ all the time, whether through query expressions or method calls with lambda expressions.

LINQ didn't just give C# developers new tools; it encouraged us to think about data transformations in a new way based on functional programming. This affects more than data access. LINQ provided the initial impetus to take on more functional ideas, but many C# developers have embraced those ideas and taken them further.

C# 4 made a radical change in terms of dynamic typing, but I don't think that affected as many developers as LINQ. Then C# 5 came along and changed the game again, this time with respect to asynchrony.

1.1.4 Asynchrony

Asynchrony has been difficult in mainstream languages for a long time. More niche languages have been created with asynchrony in mind from the start, and some functional languages have made it relatively easy as just one of the things they handle neatly. But C# 5 brought a new level of clarity to programming asynchrony in a mainstream language with a feature usually referred to as *async/await*. The feature consists of two complementary parts around `async` methods:

- Async methods produce a result representing an asynchronous operation with no effort on the part of the developer. This result type is usually Task or Task<T>.
 - Async methods use await expressions to consume asynchronous operations. If the method tries to await an operation that hasn't completed yet, the method pauses asynchronously until the operation completes and then continues.

NOTE More properly, I could call these *asynchronous functions*, because anonymous methods and lambda expressions can be asynchronous, too.

Exactly what's meant by *asynchronous operation* and *pausing asynchronously* is where things become tricky, and I won't attempt to explain this now. But the upshot is that you can write code that's asynchronous but looks mostly like the synchronous code you're more familiar with. It even allows for concurrency in a natural way. As an example, consider this asynchronous method that might be called from a Windows Forms event handler:

```
private async Task UpdateStatus()
{
    Task<Weather> weatherTask = GetWeatherAsync();
    Task<EmailStatus> emailTask = GetEmailStatusAsync();
```

Starts two operations concurrently

```

Weather weather = await weatherTask;
EmailStatus email = await emailTask;

weatherLabel.Text = weather.Description;
inboxLabel.Text = email.InboxCount.ToString();
}

```

Asynchronously waits
for them to complete

Updates the
userinterface

In addition to starting two operations concurrently and then awaiting their results, this demonstrates how `async/await` is aware of synchronization contexts. You're updating the user interface, which can be done only in a UI thread, despite also starting and waiting for long-running operations. Before `async/await`, this would've been complex and error prone.

I don't claim that `async/await` is a silver bullet for asynchrony. It doesn't magically remove all the complexity that naturally comes with the territory. Instead, it lets you focus on the inherently difficult aspects of asynchrony by taking away a lot of the boilerplate code that was previously required.

All of the features you've seen so far aim to make code simpler. The final aspect I want to mention is slightly different.

1.1.5 **Balancing efficiency and complexity**

I remember my first experiences with Java; it was entirely interpreted and painfully slow. After a while, optional just-in-time (JIT) compilers became available, and eventually it was taken almost for granted that any Java implementation would be JIT-compiled.

Making Java perform well took a lot of effort. This effort wouldn't have happened if the language had been a flop. But developers saw the potential and already felt more productive than they had before. Speed of development and delivery can often be more important than application speed.

C# was in a slightly different situation. The Common Language Runtime (CLR) was pretty efficient right from the start. The language support for easy interop with native code and for performance-sensitive unsafe code with pointers helps, too. C# performance continues to improve over time. (I note with a wry smile that Microsoft is now introducing tiered JIT compilation broadly like the Java HotSpot JIT compiler.)

But different workloads have different performance demands. As you'll see in section 1.2, C# is now in use across a surprising variety of platforms, including gaming and microservices, both of which can have difficult performance requirements.

Asynchrony helps address performance in some situations, but C# 7 is the most overtly performance-sensitive release. Read-only structs and a much larger surface area for `ref` features help to avoid redundant copying. The `Span<T>` feature present in modern frameworks and supported by `ref`-like struct types reduces unnecessary allocation and garbage collection. The hope is clearly that when used carefully, these techniques will cater to the requirements of specific developers.

I have a slight sense of unease around these features, as they still feel complex to me. I can't reason about a method using an `in` parameter as clearly as I can about

regular value parameters, and I’m sure it will take a while before I’m comfortable with what I can and can’t do with ref locals and ref returns.

My hope is that these features will be used in moderation. They’ll simplify code in situations that benefit from them, and they will no doubt be welcomed by the developers who maintain that code. I look forward to experimenting with these features in personal projects and becoming more comfortable with the balance between improved performance and increased code complexity.

I don’t want to sound this note of caution too loudly. I suspect the C# team made the right choice to include the new features regardless of how much or little I’ll use them in my work. I just want to point out that you don’t have to use a feature just because it’s there. Make your decision to opt into complexity a conscious one. Speaking of opting in, C# 7 brought a new meta-feature to the table: the use of minor version numbers for the first time since C# 1.

1.1.6 Evolution at speed: Using minor versions

The set of version numbers for C# is an odd one, and it is complicated by the fact that many developers get understandably confused between the framework and the language. (There’s no C# 3.5, for example. The .NET Framework version 3.0 shipped with C# 2, and .NET 3.5 shipped with C# 3.) C# 1 had two releases: C# 1.0 and C# 1.2. Between C# 2 and C# 6 inclusive, there were only major versions that were usually backed by a new version of Visual Studio.

C# 7 bucked that trend: there were releases of C# 7.0, C# 7.1, C# 7.2, and C# 7.3, which were all available in Visual Studio 2017. I consider it highly likely that this pattern will continue in C# 8. The aim is to allow new features to evolve quickly with user feedback. The majority of C# 7.1–7.3 features have been tweaks or extensions to the features introduced in C# 7.0.

Volatility in language features can be disconcerting, particularly in large organizations. A lot of infrastructure may need to be changed or upgraded to make sure the new language version is fully supported. A lot of developers may learn and adopt new features at different paces. If nothing else, it can be a little uncomfortable for the language to change more often than you’re used to.

For this reason, the C# compiler defaults to using the earliest minor version of the latest major version it supports. If you use a C# 7 compiler and don’t specify any language version, it will restrict you to C# 7.0 by default. If you want to use a later minor version, you need to specify that in your project file and opt into the new features. You can do this in two ways, although they have the same effect. You can edit your project file directly to add a `<LangVersion>latest</LangVersion>` element in a `<PropertyGroup>`, like this:

```
<PropertyGroup>
  ...
  <LangVersion>latest</LangVersion>
</PropertyGroup>
```

Other properties

Specifies the language version of the project

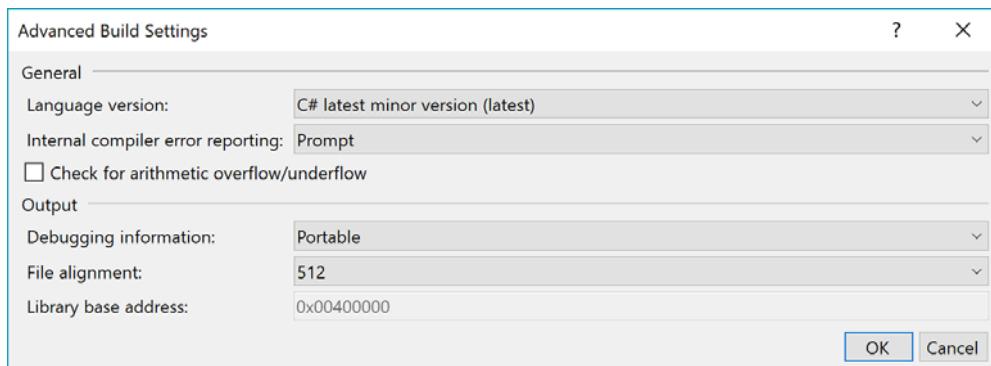


Figure 1.1 Language version settings in Visual Studio

If you don't like editing project files directly, you can go to the project properties in Visual Studio, select the Build tab, and then click the Advanced button at the bottom right. The Advanced Build Settings dialog box, shown in figure 1.1, will open to allow you to select the language version you wish to use and other options.

This option in the dialog box isn't new, but you're more likely to want to use it now than in previous versions. The values you can select are as follows:

- *default*—The first release of the latest major version
- *latest*—The latest version
- *A specific version number*—For example, 7.0 or 7.3

This doesn't change the version of the compiler you run; it changes the set of language features available to you. If you try to use something that isn't available in the version you're targeting, the compiler error message will usually explain which version is required for that feature. If you try to use a language feature that's entirely unknown to the compiler (using C# 7 features with a C# 6 compiler, for example), the error message is usually less clear.

C# as a language has come a long way since its first release. What about the platform it runs on?

1.2 An evolving platform

The last few years have been exhilarating for .NET developers. A certain amount of frustration exists as well, as both Microsoft and the .NET community come to terms with the implications of a more open development model. But the overall result of the hard work by so many people is remarkable.

For many years, running C# code would almost always mean running on Windows. It would usually mean either a client-side app written in Windows Forms or Windows Presentation Foundation (WPF) or a server-side app written with ASP.NET and probably running behind Internet Information Server (IIS). Other options have been

available for a long time, and the Mono project in particular has a rich history, but the mainstream of .NET development was still on Windows.

As I write this in June 2018, the .NET world is very different. The most prominent development is .NET Core, a runtime and framework that is portable and open source, is fully supported by Microsoft on multiple operating systems, and has streamlined development tooling. Only a few years ago, that would've been unthinkable. Add to that a portable and open source IDE in the form of Visual Studio Code, and you get a flourishing .NET ecosystem with developers working on all kinds of local platforms and then deploying to all kinds of server platforms.

It would be a mistake to focus too heavily on .NET Core and ignore the many other ways C# runs these days. Xamarin provides a rich multiplatform mobile experience. Its GUI framework (Xamarin Forms) allows developers to create user interfaces that are fairly uniform across different devices where that's appropriate but that can take advantage of the underlying platform, too.

Unity is one of the most popular game-development platforms in the world. With a customized Mono runtime and ahead-of-time compilation, it can provide challenges to C# developers who are used to more-traditional runtime environments. But for many developers, this is their first or perhaps their only experience with the language.

These widely adopted platforms are far from the only ones making C#. I've recently been working with Try .NET and Blazor for very different forms of browser/C# interaction.

Try .NET allows users to write code in a browser, with autocompletion, and then build and run that code. It's great for experimenting with C# with a barrier to entry that's about as low as it can be.

Blazor is a platform for running Razor pages directly in a browser. These aren't pages rendered by a server and then displayed in the browser; the user-interface code runs within the browser using a version of the Mono runtime converted into WebAssembly. The idea of a whole runtime executing Intermediate Language (IL) via the JavaScript engine in a browser, not only on full computers but also on mobile phones, would've struck me as absurd just a few years ago. I'm glad other developers have more imagination. A lot of the innovation in this space has been made possible only by a more collaborative and open community than ever before.

1.3 **An evolving community**

I've been involved in the C# community since the C# 1.0 days, and I've never seen it as vibrant as it is today. When I started using C#, it was very much seen as an "enterprise" programming language, and there was relatively little sense of fun and exploration.¹ With that background, the open source C# ecosystem grew fairly slowly compared with other languages, including Java, which was also considered an enterprise

¹ Don't get me wrong; it was a pleasant community to be part of, and there have always been people experimenting with C# for fun.

language. Around the time of C# 3, the alt.NET community was looking beyond the mainstream of .NET development, and this was seen as being against Microsoft in some senses.

In 2010, the NuGet (initially NuPack) package manager was launched, which made it much easier to produce and consume class libraries, whether commercial or open source. Even though the barrier of downloading a zip file, copying a DLL into somewhere appropriate, and then adding a reference to it doesn't sound hugely significant, every point of friction can put developers off.

NOTE Package managers other than NuGet were developed even earlier, and the OpenWrap project developed by Sébastien Lambotte was particularly influential.

Fast-forward to 2014, and Microsoft announced that its Roslyn compiler platform was going to become open source under the umbrella of the new .NET Foundation. Then .NET Core was announced under the initial codename Project K; DNX came later, followed by the .NET Core tooling that's now released and stable. Then came ASP.NET Core. And Entity Framework Core. And Visual Studio Code. The list of products that truly live and breathe on GitHub goes on.

The technology has been important, but the new embrace of open source by Microsoft has been equally vital for a healthy community. Third-party open source packages have blossomed, including innovative uses for Roslyn and integrations within .NET Core tooling that just feel right.

None of this has happened in a vacuum. The rise of cloud computing makes .NET Core even more important to the .NET ecosystem than it would've been otherwise; support for Linux isn't optional. But because .NET Core is available, there's now nothing special about packaging up an ASP.NET Core service in a Docker image, deploying it with Kubernetes, and using it as just one part of a larger application that could involve many languages. The cross-pollination of good ideas between many communities has always been present, but it is stronger than ever right now.

You can learn C# in a browser. You can run C# anywhere. You can ask questions about C# on Stack Overflow and myriad other sites. You can join in the discussion about the future of the language on the C# team's GitHub repository. It's not perfect; we still have collective work to do in order to make the C# community as welcoming as it possibly can be for everyone, but we're in a great place already.

I'd like to think that *C# in Depth* has its own small place in the C# community, too. How has this book evolved?

1.4 An evolving book

You're reading the fourth edition of *C# in Depth*. Although the book hasn't evolved at the same pace as the language, platform, or community, it also has changed. This section will help you understand what is covered in this book.

1.4.1 Mixed-level coverage

The first edition of *C# in Depth* came out in April 2008, which was coincidentally the same time that I joined Google. Back then, I was aware that a lot of developers knew C# 1 fairly well, but they were picking up C# 2 and C# 3 as they went along without a firm grasp of how all the pieces fit together. I aimed to address that gap by diving into the language at a depth that would help readers understand not only what each feature did but why it was designed that way.

Over time, the needs of developers change. It seems to me that the community has absorbed a deeper understanding of the language almost by osmosis, at least for earlier versions. Attaining deeper understanding of the language won't be a universal experience, but for the fourth edition, I wanted the emphasis to be on the newer versions. I still think it's useful to understand the evolution of the language version by version, but there's less need to look at every detail of the features in C# 2–4.

NOTE Looking at the language one version at a time isn't the best way to learn the language from scratch, but it's useful if you want to understand it deeply. I wouldn't use the same structure to write a book for C# beginners.

I'm also not keen on thick books. I don't want *C# in Depth* to be intimidating, hard to hold, or hard to write in. Keeping 400 pages of coverage for C# 2–4 just didn't feel right. For that reason, I've compressed my coverage of those versions. Every feature is mentioned, and I go into detail where I feel it's appropriate, but there's less depth than in the third edition. Use the coverage in the fourth edition as a review of topics you already know and to help you determine topics you want to read more about in the third edition. You can find a link to access an electronic copy of the third edition at www.manning.com/books/c-sharp-in-depth-fourth-edition. Versions 5–7 of the language are covered in more detail in this edition. Asynchrony is still a tough topic to understand, and the third edition obviously doesn't cover C# 6 or 7 at all.

Writing, like software engineering, is often a balancing act. I hope the balance I've struck between detail and brevity works for you.

TIP If you have a physical copy of this book, I strongly encourage you to write in it. Make note of places where you disagree or parts that are particularly useful. The act of doing this will reinforce the content in your memory, and the notes will serve as reminders later.

1.4.2 Examples using Noda Time

Most of the examples I provide in the book are standalone. But to make a more compelling case for some features, it's useful to be able to point to where I use them in production code. Most of the time, I use Noda Time for this.

Noda Time is an open source project I started in 2009 to provide a better date and time library for .NET. It serves a secondary purpose, though: it's a great sandbox

project for me. It helps me hone my API design skills, learn more about performance and benchmarking, and test new C# features. All of this without breaking users, of course.

Every new version of C# has introduced features that I've been able to use in Noda Time, so I think it makes sense to use those as concrete examples in this book. All of the code is available on GitHub, which means you can clone it and experiment for yourself. The purpose of using Noda Time in examples isn't to persuade you to use the library, but I'm not going to complain if that happens to be a side effect.

In the rest of the book, I'll assume that you know what I'm talking about when I refer to Noda Time. In terms of making it suitable for examples, the important aspects of it are as follows:

- The code needs to be as readable as possible. If a language feature lets me refactor for readability, I'll jump at the chance.
- Noda Time follows semantic versioning, and new major versions are rare. I pay attention to the backward-compatibility aspects of applying new language features.
- I don't have concrete performance goals, because Noda Time can be used in many contexts with different requirements. I do pay attention to performance and will embrace features that improve efficiency, so long as they don't make the code much more complex.

To find out more about the project and check out its source code, visit <https://nodatime.org> or <https://github.com/nodatime/nodatime>.

1.4.3 Terminology choices

I've tried to follow the official C# terminology as closely as I can within the book, but occasionally I've allowed clarity to take precedence over precision. For example, when writing about asynchrony, I often refer to *async methods* when the same information also applies to asynchronous anonymous functions. Likewise, object initializers apply to accessible fields as well as properties, but it's simpler to mention that once and then refer only to properties within the rest of the explanation.

Sometimes the terms within the specification are rarely used in the wider community. For example, the specification has the notion of a *function member*. That's a method, property, event, indexer, user-defined operator, instance constructor, static constructor, or finalizer. It's a term for any type member that can contain executable code, and it's useful when describing language features. It's not nearly as useful when you're looking at your own code, which is why you may never have heard of it before. I've tried to use terms like this sparingly, but my view is that it's worth becoming somewhat familiar with them in the spirit of getting closer to the language.

Finally, some concepts don't have any official terminology but are still useful to refer to in a shorthand form. The one I'll use most often is probably *unspeakable names*.

This term, coined by Eric Lippert, refers to an identifier generated by the compiler to implement features such as iterator blocks or lambda expressions.² The identifier is valid for the CLR but not valid in C#; it's a name that can't be "spoken" within the language, so it's guaranteed not to clash with your code.

Summary

I love C#. It's both comfortable and exciting, and I love seeing where it's going next. I hope this chapter has passed on some of that excitement to you. But this has been only a taste. Let's get onto the real business of the book without further delay.

² We think it was Eric, anyway. Eric can't remember for sure and thinks Anders Hejlsberg may have come up with the term first. I'll always associate it with Eric, though, along with his classification for exceptions: fatal, boneheaded, vexing, or exogenous.

Part 2

C# 2–5

T

his part of the book covers all the features introduced between C# 2 (shipped with Visual Studio 2005) and C# 5 (shipped with Visual Studio 2012). This is the same set of features that took up the entire third edition of this book. Much of it feels like ancient history now; for example, we simply take it for granted that C# includes generics.

This was a tremendously productive period for C#. Some of the features I'll cover in this part are generics, nullable value types, anonymous methods, method group conversions, iterators, partial types, static classes, automatically implemented properties, implicitly typed local variables, implicitly typed arrays, object initializers, collection initializers, anonymous types, lambda expressions, extension methods, query expressions, dynamic typing, optional parameters, named arguments, COM improvements, generic covariance and contravariance, `async/await`, and caller information attributes. Phew!

I expect most of you to be at least somewhat familiar with most of the features, so I ramp up pretty fast in this part. Likewise, for the sake of reasonable brevity, I haven't gone into as much detail as I did in the third edition. The intention is to cover a variety of reader needs:

- An introduction to features you may have missed along the way
- A reminder of the features you once knew about but have forgotten
- An explanation of the reasons behind the features: why they were introduced and why they were designed in the way they were
- A quick reference in case you know what you want to do but have forgotten some syntax

If you want more detail, please refer to the third edition. As a reminder, purchase of the fourth edition entitles you to an e-book copy of the third edition.

There's one exception to this brief coverage rule: I've completely rewritten the coverage of `async/await`, which is the largest feature in C# 5. Chapter 5 covers what you need to know to use `async/await`, and chapter 6 addresses how it's implemented behind the scenes. If you're new to `async/await`, you'll almost certainly want to wait until you've used it a bit before you read chapter 6, and even then, you shouldn't expect it to be a simple read. I've tried to explain things as accessibly as I can, but the topic is fundamentally complex. I do encourage you to try, though; understanding `async/await` at a deep level can help boost your confidence when using the feature, even if you never need to dive into the IL the compiler generates for your own code. The good news is that after chapter 6, you'll find a little relief in the form of chapter 7. It's the shortest chapter in the book and a chance to recover before exploring C# 6.

With all introductions out of the way, brace yourself for an onslaught of features.

C# 2

This chapter covers

- Using generic types and methods for flexible, safe code
- Expressing the absence of information with nullable value types
- Constructing delegates relatively easily
- Implementing iterators without writing boilerplate code

If your experience with C# goes far enough back, this chapter will be a reminder of just how far we've come and a prompt to be grateful for a dedicated and smart language design team. If you've never programmed C# without generics, you may end up wondering how C# ever took off without these features.¹ Either way, you may still find features you weren't aware of or details you've never considered listed here.

It's been more than 10 years since C# 2 was released (with Visual Studio 2005), so it can be hard to get excited about features in the rearview mirror. You shouldn't

¹ For me, the answer to this one is simple: C# 1 was a more productive language for many developers than Java was at the time.

underestimate how important its release was at the time. It was also painful: the upgrade from C# 1 and .NET 1.x to C# 2 and .NET 2.0 took a long time to roll through the industry. Subsequent evolutions have been much quicker. The first feature from C# 2 is the one almost all developers consider to be the most important: generics.

2.1 Generics

Generics allow you to write general-purpose code that's type safe at compile time using the same type in multiple places without knowing what that type is beforehand. When generics were first introduced, their primary use was for collections, but in modern C# code, they crop up everywhere. They're probably most heavily used for the following:

- Collections (they're just as useful in collections as they ever were)
- Delegates, particularly in LINQ
- Asynchronous code, where a `Task<T>` is a promise of a future value of type `T`
- Nullable value types, which I'll talk about more in section 2.2

This isn't the limit of their usefulness by any means, but even those four bullets mean that C# programmers use generics on a daily basis. Collections provide the simplest way of explaining the benefits of generics, because you can look at collections in .NET 1 and compare them with the generic collections in .NET 2.

2.1.1 Introduction by example: Collections before generics

.NET 1 had three broad kinds of collections:

- *Arrays*—These have direct language and runtime support. The size is fixed at initialization.
- *Object-based collections*—Values (and keys where relevant) are described in the API by using `System.Object`. These have no collection-specific language or runtime support, although language features such as indexers and `foreach` statements can be used with them. `ArrayList` and `Hashtable` are the most commonly used examples.
- *Specialized collections*—Values are described in the API with a specific type, and the collection can be used for only that type. `StringCollection` is a collection of strings, for example; its API looks like `ArrayList` but using `String` instead of `Object` for anything referring to a value.

Arrays and specialized collections are *statically typed*, by which I mean that the API prevents you from putting the wrong kind of value in a collection, and when you fetch a value from the collection, you don't need to cast the result back to the type you expect it to be.

NOTE Reference type arrays are only *mostly* safe when storing values because of array covariance. I view array covariance as an early design mistake that's beyond the scope of this book. Eric Lippert wrote about this at <http://mng.bz/gYPv> as part of his series of blog posts on covariance and contravariance.

Let's make this concrete: suppose you want to create a collection of strings in one method (`GenerateNames`) and print those strings out in another method (`PrintNames`). You'll look at three options to keep the collection of names—arrays, `ArrayList`, and `StringCollection`—and weigh the pros and cons of each. The code looks similar in each case (particularly for `PrintNames`), but bear with me. We'll start with arrays.

Listing 2.1 Generating and printing names by using arrays

```
static string[] GenerateNames()
{
    string[] names = new string[4];
    names[0] = "Gamma";
    names[1] = "Vlissides";
    names[2] = "Johnson";
    names[3] = "Helm";
    return names;
}

static void PrintNames(string[] names)
{
    foreach (string name in names)
    {
        Console.WriteLine(name);
    }
}
```

← Size of array needs to be known at creation time

I haven't used an array initializer here, because I want to mimic the situation where the names are discovered only one at a time, such as when reading them from a file. Notice that you need to allocate the array to be the right size to start with, though. If you really were reading from a file, you'd either need to find out how many names there were before you started, or you'd need to write more-complicated code. For example, you could allocate one array to start with, copy the contents to a larger array if the first one filled up, and so on. You'd then need to consider creating a final array of just the right size if you ended up with an array larger than the exact number of names.

The code used to keep track of the size of our collection so far, reallocate an array, and so on is repetitive and can be encapsulated in a type. As it happens, that's just what `ArrayList` does.

Listing 2.2 Generating and printing names by using ArrayList

```
static ArrayList GenerateNames()
{
    ArrayList names = new ArrayList();
    names.Add("Gamma");
    names.Add("Vlissides");
    names.Add("Johnson");
    names.Add("Helm");
    return names;
}
```

```

static void PrintNames(ArrayList names)
{
    foreach (string name in names)      ← | What happens if the ArrayList
    {                                     contains a nonstring?
        Console.WriteLine(name);
    }
}

```

That's cleaner in terms of our `GenerateNames` method: you don't need to know how many names you have before you start adding to the collection. But equally, there's nothing to stop you from adding a nonstring to the collection; the type of the `ArrayList.Add` parameter is just `Object`.

Furthermore, although the `PrintNames` method looks safe in terms of types, it's not. The collection can contain any kind of object reference. What would you expect to happen if you added a completely different type (a `WebRequest`, as an odd example) to the collection, and then tried to print it? The `foreach` loop hides an implicit cast, from `object` to `string`, because of the type of the `name` variable. That cast can fail in the normal way with an `InvalidCastException`. Therefore, you've fixed one problem but caused another. Is there anything that solves both of these?

Listing 2.3 Generating and printing names by using `StringCollection`

```

static StringCollection GenerateNames()
{
    StringCollection names = new StringCollection();
    names.Add("Gamma");
    names.Add("Vlissides");
    names.Add("Johnson");
    names.Add("Helm");
    return names;
}

static void PrintNames(StringCollection names)
{
    foreach (string name in names)
    {
        Console.WriteLine(name);
    }
}

```

Listing 2.3 is identical to listing 2.2 except for replacing `ArrayList` with `StringCollection` everywhere. That's the whole point of `StringCollection`: it should feel like a pleasant general-purpose collection but specialized to only handle strings. The parameter type of `StringCollection.Add` is `String`, so you can't add a `WebRequest` to it through some odd bug in our code. The resulting effect is that when you print the names, you can be confident that the `foreach` loop won't encounter any nonstring references. (You could still see a null reference, admittedly.)

That's great if you always need only strings. But if you need a collection of some other type, you have to either hope that there's already a suitable collection type in

the framework or write one yourself. This was such a common task that there's a `System.Collections.CollectionBase` abstract class to make the work somewhat less repetitive. There are also code generators to avoid having to write it all by hand.

That solves both problems from the previous solution, but the cost of having all these extra types around is way too high. There's a maintenance cost in keeping them up-to-date as the code generator changes. There are efficiency costs in terms of compilation time, assembly size, JITting time, and keeping the code in memory. Most important, there's a human cost in keeping track of all the collection classes available.

Even if those costs weren't too high, you'd be missing the ability to write a method that can work on any collection type in a statically typed way, potentially using the collection's element type in another parameter or in the return type. For example, say you want to write a method to create a copy of the first N elements of a collection into a new one, which was then returned. You could write a method that returns an `ArrayList`, but that loses the goodness of static typing. If you pass in a `StringCollection`, you'd want a `StringCollection` back. The string aspect is part of the input to the method, which then needs to be propagated to the output as well. You had no way of expressing that in the language when using C# 1. Enter generics.

2.1.2 Generics save the day

Let's get straight to the solution for our `GenerateNames/PrintNames` code and use the `List<T>` generic type. `List<T>` is a collection in which `T` is the element type of the collection—string, in our case. You can replace `StringCollection` with `List<string>` everywhere.²

Listing 2.4 Generating and printing names with `List<T>`

```
static List<string> GenerateNames()
{
    List<string> names = new List<string>();
    names.Add("Gamma");
    names.Add("Vlissides");
    names.Add("Johnson");
    names.Add("Helm");
    return names;
}

static void PrintNames(List<string> names)
{
    foreach (string name in names)
    {
        Console.WriteLine(name);
    }
}
```

² I'm deliberately not going into the possibility of using interfaces for return types and parameters. That's an interesting topic, but I don't want to distract you from generics.

`List<T>` solves all the problems we talked about before:

- You don't need to know the size of the collection beforehand, unlike with arrays.
- The exposed API uses `T` everywhere it needs to refer to the element type, so you know that a `List<string>` will contain only string references. You'll get a compile-time error if you try to add anything else, unlike with `ArrayList`.
- You can use it with any element type without worrying about generating code and managing the result, unlike with `StringCollection` and similar types.

Generics also solve the problem of expressing an element type as an input to a method. To delve into that aspect more deeply, you'll need more terminology.

TYPE PARAMETERS AND TYPE ARGUMENTS

The terms *parameter* and *argument* predate generics in C# and have been used in other languages for decades. A method declares its inputs as parameters, and they're provided by calling code in the form of arguments. Figure 2.1 shows how the two relate to each other.

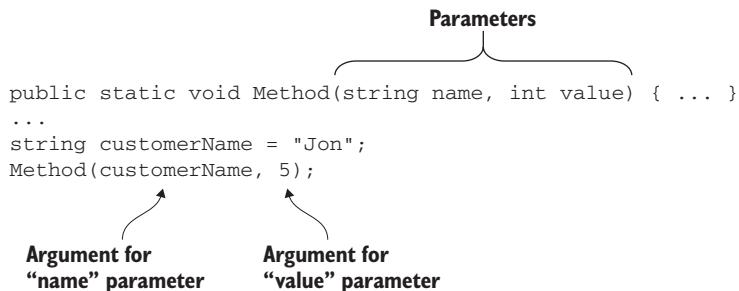


Figure 2.1 Relationship between method parameters and arguments

The values of the arguments are used as the initial values for the parameters within the method. In generics, you have *type parameters* and *type arguments*, which are the same idea but applied to types. The declaration of a generic type or method includes type parameters in angle brackets after the name. Within the body of the declaration, the code can use the type parameter as a normal type (just one it doesn't know much about).

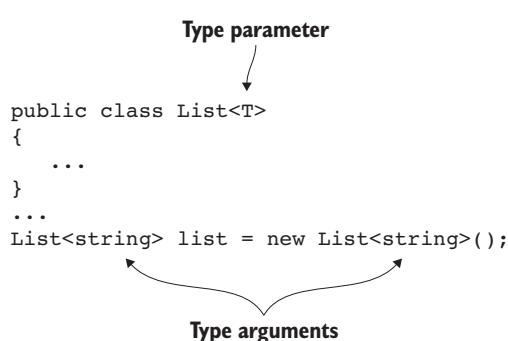


Figure 2.2 Relationship between type parameters and type arguments

The code using the generic type or method then specifies the type arguments in angle brackets after the name as well. Figure 2.2 shows this relationship in the context of `List<T>`.

The code using the generic type or method then specifies the type arguments in angle brackets after the name as well. Figure 2.2 shows this relationship in the context of `List<T>`.

Now imagine the complete API of `List<T>`: all the method signatures, properties, and so on. If you’re using the `list` variable shown in the figure, any `T` that appears in the API becomes `string`. For example, the `Add` method in `List<T>` has the following signature:

```
public void Add(T item)
```

But if you type `list.Add()` into Visual Studio, IntelliSense will prompt you as if the `item` parameter had been declared with a type of `string`. If you try to pass in an argument of another type, it will result in a compile-time error.

Although figure 2.2 refers to a generic class, methods can be generic as well. The method declares type parameters, and those type parameters can be used within other parts of the method signature. Method type parameters are often used as type arguments to other types within the signature. The following listing shows a solution to the method you couldn’t implement earlier: something to create a new collection containing the first N elements of an existing one but in a statically typed way.

Listing 2.5 Copying elements from one collection to another

```
public static List<T> CopyAtMost<T>(
    List<T> input, int maxElements)
{
    int actualCount = Math.Min(input.Count, maxElements);
    List<T> ret = new List<T>(actualCount);
    for (int i = 0; i < actualCount; i++)
    {
        ret.Add(input[i]);
    }
    return ret;
}

static void Main()
{
    List<int> numbers = new List<int>();
    numbers.Add(5);
    numbers.Add(10);
    numbers.Add(20);
    List<int> firstTwo = CopyAtMost<int>(numbers, 2);
    Console.WriteLine(firstTwo.Count);
}
```

Method declares a type parameter T and uses it in parameters and return type.

Type parameter used in method body

Call to method using int as the type parameter

←

Plenty of generic methods use the type parameter only once in the signature³ and without it being a type argument to any generic types. But the ability to use a type parameter to express a relationship between the types of regular parameters and the return type is a huge part of the power of generics.

³ Although it’s valid to write a generic method that doesn’t use the type parameter anywhere else in the signature, that’s rarely useful.

Likewise, generic types can use their type parameters as type arguments when declaring a base class or an implemented interface. For example, the `List<T>` type implements the `IEnumerable<T>` interface, so the class declaration could be written like this:

```
public class List<T> : IEnumerable<T>
```

NOTE In reality, `List<T>` implements multiple interfaces; this is a simplified form.

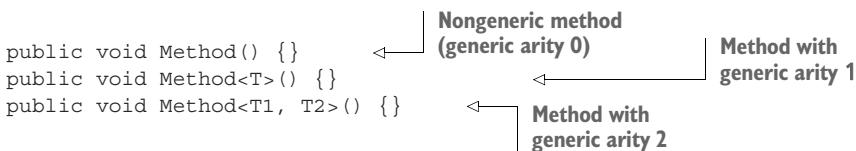
ARITY OF GENERIC TYPES AND METHODS

Generic types or methods can declare multiple type parameters by separating them with commas within the angle brackets. For example, the generic equivalent of the .NET 1 `Hashtable` class is declared like this:

```
public class Dictionary<TKey, TValue>
```

The generic *arity* of a declaration is the number of type parameters it has. To be honest, this is a term that's more useful to authors than in everyday usage when writing code, but I'd argue it's still worth knowing. You can think of a nongeneric declaration as one with generic arity 0.

The generic arity of a declaration is effectively part of what makes it unique. As an example, I've already referred to the `IEnumerable<T>` interface introduced in .NET 2.0, but that's a distinct type from the nongeneric `IEnumerable` interface that was already part of .NET 1.0. Likewise, you can write methods with the same name but a different generic arity, even if their signatures are otherwise the same:



When declaring types with different generic arity, the types don't have to be of the same kind, although they usually are. As an extreme example, consider these type declarations that can all coexist in one highly confusing assembly:

```
public enum IAmConfusing {}
public class IAmConfusing<T> {}
public struct IAmConfusing<T1, T2> {}
public delegate void IAmConfusing<T1, T2, T3> {}
public interface IAmConfusing<T1, T2, T3, T4> {}
```

Although I'd strongly discourage code like the above, one reasonably common pattern is to have a nongeneric static class providing helper methods that refer to other generic types with the same name (see section 2.5.2 for more about static classes). For example, you'll see the `Tuple` class in section 2.1.4, which is used to create instances of the various generic `Tuple` classes.

Just as multiple types can have the same name but different generic arity, so can generic methods. It's like creating overloads based on the parameters, except this is overloading based on the number of type parameters. Note that although the generic arity keeps declarations separate, type parameter names don't. For example, you can't declare two methods like this:

```
public void Method<TFirst>() {}           ← Compile-time error; can't overload
public void Method<TSecond>() {}             solely by type parameter name
```

These are deemed to have equivalent signatures, so they aren't permitted under the normal rules of method overloading. You can write method overloads that use different type parameter names so long as the methods differ in other ways (such as the number of regular parameters), although I can't remember ever wanting to do so.

While we're on the subject of multiple type parameters, you can't give two type parameters in the same declaration the same name just like you can't declare two regular parameters the same name. For example, you can't declare a method like this:

```
public void Method<T, T>() {}           ← Compile-time error;
                                         duplicate type
                                         parameter T
```

It's fine for two type arguments to be the same, though, and that's often what you want. For example, to create a string-to-string mapping, you might use a `Dictionary<string, string>`.

The earlier example of `IAmConfusing` used an enum as the nongeneric type. That was no coincidence, because I wanted to use it to demonstrate my next point.

2.1.3 What can be generic?

Not all types or type members can be generic. For types, it's reasonably simple, partly because relatively few kinds of types can be declared. Enums can't be generic, but classes, structs, interfaces, and delegates all can be.

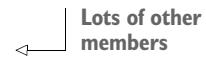
For type members, it's slightly more confusing; some members may look like they're generic because they use other generic types. Remember that a declaration is generic only if it introduces new type parameters.

Methods and nested types can be generic, but all of the following have to be non-generic:

- Fields
- Properties
- Indexers
- Constructors
- Events
- Finalizers

As an example of how you might be tempted to think of a field as being generic even though it's not, consider this generic class:

```
public class ValidatingList<TItem>
{
    private readonly List<TItem> items = new List<TItem>();
```



I've named the type parameter `TItem` simply to differentiate it from the `T` type parameter of `List<T>`. Here, the `items` field is of type `List<TItem>`. It uses the type parameter `TItem` as a type argument for `List<T>`, but that's a type parameter introduced by the class declaration, not by the field declaration.

For most of these, it's hard to conceive how the member could be generic. Occasionally, I've wanted to write a generic constructor or indexer, though, and the answer is almost always to write a generic method instead.

Speaking of generic methods, I gave only a simplified description of type arguments earlier when I was describing the way generic methods are called. In some cases, the compiler can determine the type arguments for a call without you having to provide them in the source code.

2.1.4 Type inference for type arguments to methods

Let's look back at the crucial parts of listing 2.5. You have a generic method declared like this:

```
public static List<T> CopyAtMost<T>(List<T> input, int maxElements)
```

Then, in the `Main` method, you declare a variable of type `List<int>` and later use that as an argument to the method:

```
List<int> numbers = new List<int>();
...
List<int> firstTwo = CopyAtMost<int>(numbers, 2);
```

I've highlighted the method call here. You need a type argument to the `CopyAtMost` call, because it has a type parameter. But you don't have to specify that type argument in the source code. You can rewrite that code as follows:

```
List<int> numbers = new List<int>();
...
List<int> firstTwo = CopyAtMost(numbers, 2);
```

This is exactly the same method call in terms of the IL the compiler will generate. But you haven't had to specify the type argument of `int`; the compiler inferred that for you. It did that based on your argument for the first parameter in the method. You're using an argument of type `List<int>` as the value for a parameter of type `List<T>`, so `T` has to be `int`.

Type inference can use only the arguments you pass to a method, not what you do with the result. It also has to be complete; you either explicitly specify all the type arguments or none of them.

Although type inference applies only to methods, it can be used to more easily construct instances of generic types. For example, consider the `Tuple` family of types introduced in .NET 4.0. This consists of a nongeneric static `Tuple` class and multiple generic classes: `Tuple<T1>`, `Tuple<T1, T2>`, `Tuple<T1, T2, T3>`, and so forth. The static class has a set of overloaded `Create` factory methods like this:

```
public static Tuple<T1> Create<T1>(T1 item1)
{
    return new Tuple<T1>(item1);
}

public static Tuple<T1, T2> Create<T1, T2>(T1 item1, T2 item2)
{
    return new Tuple<T1, T2>(item1, item2);
}
```

These look pointlessly trivial, but they allow type inference to be used where otherwise the type arguments would have to be explicitly specified when creating tuples. Instead of this

```
new Tuple<int, string, int>(10, "x", 20)
```

you can write this:

```
Tuple.Create(10, "x", 20)
```

This is a powerful technique to be aware of; it's generally simple to implement and can make working with generic code a lot more pleasant.

I'm not going to go into the details of how generic type inference works. It's changed a lot over time as the language designers figure out ways of making it work in more cases. Overload resolution and type inference are closely tied together, and they intersect with all kinds of other features (such as inheritance, conversions, and optional parameters in C# 4). This is the area of the specification I find the most complex,⁴ and I couldn't do it justice here.

Fortunately, this is one area where understanding the details wouldn't help very much in day-to-day coding. In any particular situation, three possibilities exist:

- Type inference succeeds and gives you the result you want. Hooray.
- Type inference succeeds but gives you a result you didn't want. Just explicitly specify type arguments or cast some of the arguments. For example, if you wanted a `Tuple<int, object, int>` from the preceding `Tuple.Create` call,

⁴ I'm not alone in this. At the time of this writing, the spec for overload resolution is broken. Efforts to fix it for the C# 5 ECMA standard failed; we're going to try again for the next edition.

you could specify the type arguments to `Tuple.Create` explicitly or just call `new Tuple<int, object, int>(...)` or call `Tuple.Create(10, (object) "x", 20)`.

- Type inference fails at compile time. Sometimes this can be fixed by casting some of your arguments. For example, the `null` literal doesn't have a type, so type inference will fail for `Tuple.Create(null, 50)` but succeed for `Tuple.Create((string) null, 50)`. Other times you just need to explicitly specify the type arguments.

For the last two cases, the option you pick rarely makes much difference to readability in my experience. Understanding the details of type inference can make it easier to predict what will work and what won't, but it's unlikely to repay the time invested in studying the specification. If you're curious, I'd never actively discourage anyone from reading the specification. Just don't be surprised when you find it alternates between feeling like a maze of twisty little passages, all alike, and a maze of twisty little passages, all different.

This alarmist talk of complicated language details shouldn't detract from the convenience of type inference, though. C# is considerably easier to use because of its presence.

So far, all the type parameters we've talked about have been unconstrained. They could stand in for any type. That's not always what you want, though; sometimes, you want only certain types to be used as type arguments for a particular type parameter. That's where type constraints come in.

2.1.5 Type constraints

When a type parameter is declared by a generic type or method, it can also specify *type constraints* that restrict which types can be provided as type arguments. Suppose you want to write a method that formats a list of items and ensures that you format them in a particular culture instead of the default culture of the thread. The `IFormattable` interface provides a suitable `ToString(string, IFormatProvider)` method, but how can you make sure you have an appropriate list? You might expect a signature like this:

```
static void PrintItems(List<IFormattable> items)
```

But that would hardly ever be useful. You couldn't pass a `List<decimal>` to it, for example, even though `decimal` implements `IFormattable`; a `List<decimal>` isn't convertible to `List<IFormattable>`.

NOTE We'll go into the reasons for this more deeply in chapter 4, when we consider generic variance. For the moment, just treat this as a simple example for constraints.

What you need to express is that the parameter is a list of some element type, where the element type implements the `IFormattable` interface. The "some element type" part suggests that you might want to make the method generic, and "where the

element type implements the `IFormattable` interface” is precisely the ability that type constraints give us. You add a `where` clause at the end of the method declaration, like this:

```
static void PrintItems<T>(List<T> items) where T : IFormattable
```

The way you’ve constrained `T` here doesn’t just change which values can be passed to the method; it also changes what you can do with a value of type `T` within the method. The compiler knows that `T` implements `IFormattable`, so it allows the `IFormattable.ToString(string, IFormatProvider)` method to be called on any `T` value.

Listing 2.6 Printing items in the invariant culture by using type constraints

```
static void PrintItems<T>(List<T> items) where T : IFormattable
{
    CultureInfo culture = CultureInfo.InvariantCulture;
    foreach (T item in items)
    {
        Console.WriteLine(item.ToString(null, culture));
    }
}
```

Without the type constraints, that `ToString` call wouldn’t compile; the only `ToString` method the compiler would know about for `T` is the one declared in `System.Object`.

Type constraints aren’t limited to interfaces. The following type constraints are available:

- *Reference type constraint*—`where T : class`. The type argument must be a reference type. (Don’t be fooled by the use of the `class` keyword; it can be any reference type, including interfaces and delegates.)
- *Value type constraint*—`where T : struct`. The type argument must be a non-nullable value type (either a `struct` or an `enum`). Nullable value types (described in section 2.2) don’t meet this constraint.
- *Constructor constraint*—`where T : new()`. The type argument must have a public parameterless constructor. This enables the use of `new T()` within the body of the code to construct a new instance of `T`.
- *Conversion constraint*—`where T : SomeType`. Here, `SomeType` can be a class, an interface, or another type parameter as shown here:
 - `where T : Control`
 - `where T : IFormattable`
 - `where T1 : T2`

Moderately complex rules indicate how constraints can be combined. In general, the compiler error message makes it obvious what’s wrong when you break these rules.

One interesting and reasonably common form of constraint uses the type parameter in the constraint itself:

```
public void Sort(List<T> items) where T : IComparable<T>
```

The constraint uses `T` as the type argument to the generic `IComparable<T>` interface. This allows our sorting method to compare elements from the `items` parameter pairwise using the `CompareTo` method from `IComparable<T>`:

```
T first = ...;
T second = ...;
int comparison = first.CompareTo(second);
```

I've used interface-based type constraints more than any other kind, although I suspect what you use depends greatly on the kind of code you're writing.

When multiple type parameters exist in a generic declaration, each type parameter can have an entirely different set of constraints as in the following example:

```
 TResult Method<TArg, TResult>(TArg input)
    where TArg : IComparable<TArg>
    where TResult : class, new()
```

The diagram shows a code snippet for a generic method `Method`. It has two type parameters: `TArg` and `TResult`. There are three annotations with arrows pointing to specific parts of the code:

- An annotation for `TArg` points to the constraint `where TArg : IComparable<TArg>` with the text "TArg must implement IComparable<TArg>".
- An annotation for `TResult` points to the constraint `where TResult : class, new()` with the text "TResult must be a reference type with a parameterless constructor."
- A general annotation for the method signature points to the entire line `TResult Method<TArg, TResult>(TArg input)` with the text "Generic method with two type parameters, TArg and TResult".

We've nearly finished our whirlwind tour of generics, but I have a couple of topics left to describe. I'll start with the two type-related operators available in C# 2.

2.1.6 The `default` and `typeof` operators

C# 1 already had the `typeof()` operator accepting a type name as its only operand. C# 2 added the `default()` operator and expanded the use of `typeof` slightly.

The `default` operator is easily described. The operand is the name of a type or type parameter, and the result is the default value for that type—the same value you'd get if you declared a field and didn't immediately assign a value to it. For reference types, that's a null reference; for non-nullable value types, it's the “all zeroes” value (0, 0.0, 0.0m, false, the UTF-16 code unit with a numerical value of 0, and so on); and for nullable value types, it's the null value for the type.

The `default` operator can be used with type parameters and with generic types with appropriate type arguments supplied (where those arguments can be type parameters, too). For example, in a generic method declaring a type parameter `T`, all of these are valid:

- `default(T)`
- `default(int)`
- `default(string)`
- `default(List<T>)`
- `default(List<List<string>>)`

The type of the `default` operator is the type that's named inside it. It's most frequently used with generic type parameters, because otherwise you can usually specify the default value in a different way. For example, you might want to use the default value as the initial value for a local variable that may or may not be assigned a different value later. To make this concrete, here's a simplistic implementation of a method that may be familiar to you:

```
public T LastOrDefault(IEnumerable<T> source)
{
    T ret = default(T);           ← Declare a local variable and
    foreach (T item in source)   | assign the default value of T to it.
    {
        ret = item;             ← Replace the local variable value with
    }                           | the current one in the sequence.
    return ret;                 ← Return the
}                           | last-assigned value.
```

The `typeof` operator is slightly more complex. There are four broad cases to consider:

- No generics involved at all; for example, `typeof(string)`
- Generics involved but no type parameters; for example, `typeof(List<int>)`
- Just a type parameter; for example, `typeof(T)`
- Generics involved using a type parameter in the operand; for example, `typeof(List<TItem>)` within a generic method declaring a type parameter called `TItem`
- Generics involved but no type arguments specified in the operand; for example, `typeof(List<>)`

The first of these is simple and hasn't changed at all. All the others need a little more care, and the last introduces a new kind of syntax. The `typeof` operator is still defined to return a `Type` value, so what should it return in each of these cases? The `Type` class was augmented to know about generics. There are multiple situations to be considered; the following are a few examples:

- If you list the types within the assembly containing `List<T>`, for example, you'd expect to get `List<T>` without any specific type argument for `T`. It's a *generic type definition*.
- If you call `GetType()` on a `List<int>` object, you'd want to get a type that has the information about the type argument.
- If you ask for the base type of the generic type definition of a class declared as
`class StringDictionary<T> : Dictionary<string, T>`
you'd end up with a type with one "concrete" type argument (`string`, for the `TKey` type parameter of `Dictionary<TKey, TValue>`) and one type argument that's still a type parameter (`T`, for the `TValue` type parameter).

Frankly, it's all very confusing, but that's inherent in the problem domain. Lots of methods and properties in `Type` let you go from a generic type definition to a type with all the type arguments provided, or vice versa, for example.

Let's come back to the `typeof` operator. The simplest example to understand is `typeof(List<int>)`. That returns the Type representing `List<T>` with a type argument of `int` just as if you'd called `new List<int>().GetType()`.

The next case, `typeof(T)`, returns whatever the type argument for `T` is at that point in the code. This will always be a *closed, constructed type*, which is the specification's way of saying it's a real type with no type parameters involved anywhere. Although in most places I try to explain terminology thoroughly, the terminology around generics (open, closed, constructed, bound, unbound) is confusing and almost never useful in real life. We'll need to talk about closed, constructed types later, but I won't touch on the rest.

It's easiest to demonstrate what I mean about `typeof(T)`, and you can look at `typeof(List<T>)` in the same example. The following listing declares a generic method that prints the result of both `typeof(T)` and `typeof(List<T>)` to the console and then calls that method with two different type arguments.

Listing 2.7 Printing the result of the `typeof` operator

```
static void PrintType<T>()
{
    Console.WriteLine("typeof(T) = {0}", typeof(T));
    Console.WriteLine("typeof(List<T>) = {0}", typeof(List<T>));
}

static void Main()
{
    PrintType<string>();           ← Calls the method with a
    PrintType<int>();             ← Calls the method with
}                                     ← Prints both typeof(T)
                                         and typeof(List<T>)
```

The result of listing 2.7 is shown here:

```
typeof(T) = System.String
typeof(List<T>) = System.Collections.Generic.List`1[System.String]
typeof(T) = System.Int32
typeof(List<T>) = System.Collections.Generic.List`1[System.Int32]
```

The important point is that when you're running in a context where the type argument for `T` is `string` (during the first call), the result of `typeof(T)` is the same as `typeof(string)`. Likewise, the result of `typeof(List<T>)` is the same as the result of `typeof(List<string>)`. When you call the method again with `int` as the type argument, you get the same results as for `typeof(int)` and `typeof(List<int>)`. Whenever code is executing within a generic type or method, the type parameter always refers to a closed, constructed type.

Another takeaway from this output is the format of the name of a generic type when you're using reflection. The `List`1` indicates that this is a generic type called `List` with generic arity 1 (one type parameter), and the type arguments are shown in square brackets afterward.

The final bullet in our earlier list was `typeof(List<>)`. That appears to be missing a type argument altogether. This syntax is valid only in the `typeof` operator and refers to the generic type definition. The syntax for types with generic arity 1 is just `TypeName<>`; for each additional type parameter, you add a comma within the angle brackets. To get the generic type definition for `Dictionary< TKey, TValue >`, you'd use `typeof(Dictionary<, >)`. To get the definition for `Tuple< T1, T2, T3 >`, you'd use `typeof(Tuple<, , >)`.

Understanding the difference between a generic type definition and a closed, constructed type is crucial for our final topic: how types are initialized and how type-wide (static) state is handled.

2.1.7 Generic type initialization and state

As you saw when using the `typeof` operator, `List<int>` and `List<string>` are effectively different types that are constructed from the same generic type definition. That's not only true for how you use the types but also true for how types are initialized and how static fields are handled. Each closed, constructed type is initialized separately and has its own independent set of static fields. The following listing demonstrates this with a simple (and not thread-safe) generic counter.

Listing 2.8 Exploring static fields in generic types

```
class GenericCounter<T>
{
    private static int value;           ← One field per closed,
    static GenericCounter()           ← constructed type
    {
        Console.WriteLine("Initializing counter for {0}", typeof(T));
    }

    public static void Increment()
    {
        value++;
    }

    public static void Display()
    {
        Console.WriteLine("Counter for {0}: {1}", typeof(T), value);
    }
}

class GenericCounterDemo
{
    static void Main()
    {
        GenericCounter<string>.Increment();   ← Triggers initialization for
        GenericCounter<string>.Increment();   ← Triggers initialization for
        GenericCounter<string>.Display();     ← GenericCounter<string>
        GenericCounter<int>.Display();        ← Triggers initialization for
        GenericCounter<int>.Increment();     ← GenericCounter<int>
        GenericCounter<int>.Display();
    }
}
```

The output of listing 2.8 is as follows:

```
Initializing counter for System.String
Counter for System.String: 2
Initializing counter for System.Int32
Counter for System.Int32: 0
Counter for System.Int32: 1
```

There are two results to focus on in that output. First, the `GenericCounter<string>` value is independent of `GenericCounter<int>`. Second, the static constructor is run twice: once for each closed, constructed type. If you didn't have a static constructor, there would be fewer timing guarantees for exactly when each type would be initialized, but essentially you can regard `GenericCounter<string>` and `GenericCounter<int>` as independent types.

To complicate things further, generic types can be nested within other generic types. When that occurs, there's a separate type for each combination of type arguments. For example, consider classes like this:

```
class Outer<TOuter>
{
    class Inner<TInner>
    {
        static int value;
    }
}
```

Using `int` and `string` as type arguments, the following types are independent and each has its own `value` field:

- `Outer<string>.Inner<string>`
- `Outer<string>.Inner<int>`
- `Outer<int>.Inner<string>`
- `Outer<int>.Inner<int>`

In most code this occurs relatively rarely, and it's simple enough to handle when you're aware that what's important is the fully specified type, including any type arguments for both the leaf type and any enclosing types.

That's it for generics, which is by far the biggest single feature in C# 2 and a huge improvement over C# 1. Our next topic is nullable value types, which are firmly based on generics.

2.2 Nullable value types

Tony Hoare introduced null references into Algol in 1965 and has subsequently called it his “billion-dollar mistake.” Countless developers have become frustrated when their code throws `NullReferenceException` (.NET), `NullPointerException` (Java), or other equivalents. There are canonical Stack Overflow questions with hundreds of other questions pointing at them because it's such a common problem. If nullity is so bad, why was more of it introduced in C# 2 and .NET 2.0 in the form of

nullable value types? Before we look at the implementation of the feature, let's consider the problem it's trying to solve and the previous workarounds.

2.2.1 Aim: Expressing an absence of information

Sometimes it's useful to have a variable to represent some information, but that information won't be present in every situation. Here are a few simple examples:

- You're modeling a customer order, including the company's details, but the customer may not be ordering on behalf of a company.
- You're modeling a person, including their date of birth and date of death, but the person may still be alive.
- You're modeling a filter for products, including a price range, but the customer may not have specified a maximum price.

These are all one specific form of wanting to represent the absence of a value; you can have complete information but still need to model the absence. In other situations, you may have incomplete information. In the second example, you may not know the person's date of birth not because they weren't born, but because your system doesn't have that information. Sometimes you need to represent the difference between "known to be absent" and "unknown" within your data, but often just the absence of information is enough.

For reference types, you already have a way of representing an absence of information: a null reference. If you have a `Company` class and your `Order` class has a reference to the company associated with the order, you can set it to null if the customer doesn't specify a company.

For value types in C# 1, there was no equivalent. There were two common ways of representing this:

- Use a reserved value to represent missing data. For example, you might use `decimal.MaxValue` in a price filter to represent "no maximum price specified."
- Keep a separate Boolean flag to indicate whether another field has a real value or the value should be ignored. So long as you check the flag before using the other field, its value is irrelevant in the absent case.

Neither of these is ideal. The first approach reduces the set of valid values (not so bad for `decimal` but more of a problem for `byte`, where it's more likely that you need the full range). The second approach leads to a lot of tedious and repetitive logic.

More important, both are error prone. Both require you to perform a check before using the value that might or might not be valid. If you don't perform that check, your code will proceed using inappropriate data. It'll silently do the wrong thing and quite possibly propagate the mistake to other parts of the system. Silent failure is the worst kind, because it can be hard to track down and hard to undo. I prefer nice loud exceptions that stop the broken code in its tracks.

Nullable value types encapsulate the second approach shown previously: they keep an extra flag along with the value to say whether it should be used. The encapsulation

is key here; the simplest way of using the value is also a safe one because it throws an exception if you try to use it inappropriately. The consistent use of a single type to represent a possibly missing value enables the language to make our lives easier, and library authors have an idiomatic way of representing it in their API surface, too.

With that conceptual introduction out of the way, let's look at what the framework and the CLR provide in terms of nullable value types. After you've built that foundation, I'll show you the extra features C# has adopted to make it easy to work with them.

2.2.2 CLR and framework support: The Nullable<T> struct

The core of nullable value type support is the `Nullable<T>` struct. A primitive version of `Nullable<T>` would look like this:

```
public struct Nullable<T> where T : struct
{
    private readonly T value;
    private readonly bool hasValue;

    public Nullable(T value)
    {
        this.value = value;
        this.hasValue = true;
    }

    public bool HasValue { get { return hasValue; } }

    public T Value
    {
        get
        {
            if (!hasValue)
            {
                throw new InvalidOperationException();
            }
            return value;
        }
    }
}
```

As you can see, the only declared constructor sets `hasValue` to `true`, but like all structs, there's an implicit parameterless constructor that will leave `hasValue` as `false` and `value` as the default value of `T`:

```
Nullable<int> nullable = new Nullable<int>();
Console.WriteLine(nullable.HasValue);
```

The `where T : struct` constraint on `Nullable<T>` allows `T` to be any value type except another `Nullable<T>`. It works with primitive types, enums, system-provided structs, and user-defined structs. All of the following are valid:

- `Nullable<int>`
- `Nullable<FileMode>`

- `Nullable<Guid>`
- `Nullable<LocalDate>` (from Noda Time)

But the following are invalid:

- `Nullable<string>` (`string` is a reference type)
- `Nullable<int[]>` (arrays are reference types, even if the element type is a value type)
- `Nullable<ValueType>` (`ValueType` itself isn't a value type)
- `Nullable<Enum>` (`Enum` itself isn't a value type)
- `Nullable<Nullable<int>>` (`Nullable<int>` is nullable)
- `Nullable<Nullable<Nullable<int>>>` (trying to nest the nullability further doesn't help)

The type `T` is also known as the *underlying type* of `Nullable<T>`. For example, the underlying type of `Nullable<int>` is `int`.

With just this part in place and no extra CLR, framework, or language support, you can safely use type to display the maximum price filter:

```
public void DisplayMaxPrice(Nullable<decimal> maxPriceFilter)
{
    if (maxPriceFilter.HasValue)
    {
        Console.WriteLine("Maximum price: {0}", maxPriceFilter.Value);
    }
    else
    {
        Console.WriteLine("No maximum price set.");
    }
}
```

That's well-behaved code that checks before using the value, but what about poorly written code that forgets to check first or checks the wrong thing? You can't accidentally use an inappropriate value; if you try to access `maxPriceFilter.Value` when its `HasValue` property is `false`, an exception will be thrown.

NOTE I know I made this point earlier, but I think it's important enough to restate: progress doesn't come just from making it easier to write correct code; it also comes from making it harder to write broken code or making the consequences less severe.

The `Nullable<T>` struct has methods and operators available, too:

- The parameterless `GetValueOrDefault()` method will return the value in the struct or the default value for the type if `HasValue` is `false`.
- The parameterized `GetValueOrDefault(T defaultValue)` method will return the value in the struct or the specified default value if `HasValue` is `false`.
- The `Equals(object)` and `GetHashCode()` methods declared in `object` are overridden in a reasonably obvious way, first comparing the `HasValue`

properties and then comparing the `Value` properties for equality if `HasValue` is `true` for both values.

- There's an implicit conversion from `T` to `Nullable<T>`, which always succeeds and returns a value where `HasValue` is `true`. This is equivalent to calling the parameterized constructor.
- There's an explicit conversion from `Nullable<T>` to `T`, which either returns the encapsulated value (if `HasValue` is `true`) or throws an `InvalidOperationException` (if `HasValue` is `false`). This is equivalent to using the `Value` property.

I'll return to the topic of conversions when I talk about language support. So far, the only place you've seen where the CLR needs to understand `Nullable<T>` is to enforce the `struct` type constraint. Another aspect of CLR behavior is nullable-specific, though: boxing.

BOXING BEHAVIOR

`Nullable` value types behave differently than non-nullable value types when it comes to boxing. When a value of a non-nullable value type is boxed, the result is a reference to an object of a type that's the boxed form of the original type. Say, for example, you write this:

```
int x = 5;
object o = x;
```

The value of `o` is a reference to an object of type "boxed int." The difference between boxed int and `int` isn't normally visible via C#. If you call `o.GetType()`, the `Type` returned will be equal to `typeof(int)`, for example. Some other languages (such as C++/CLI) allow developers to differentiate between the original value type and its boxed equivalent.

Nullable value types have no boxed equivalent, however. The result of boxing a value of type `Nullable<T>` depends on the `HasValue` property:

- If `HasValue` is `false`, the result is a null reference.
- If `HasValue` is `true`, the result is a reference to an object of type "boxed `T`."

The following listing demonstrates both of these points.

Listing 2.9 The effects of boxing nullable value type values

```
Nullable<int> noValue = new Nullable<int>();
object noValueBoxed = noValue;
Console.WriteLine(noValueBoxed == null);
```

↳ Boxes a value where
HasValue is false


```
Nullable<int> someValue = new Nullable<int>(5);
object someValueBoxed = someValue;
Console.WriteLine(someValueBoxed.GetType());
```

↳ Prints True: the result of
boxing is a null reference.

↳ Boxes a value where
HasValue is true

↳ Prints System.Int32: the
result is a boxed int.

When you're aware of this behavior, it's almost always what you want. This has one bizarre side effect, however. The `GetType()` method declared on `System.Object` is nonvirtual, and the somewhat complex rules around when boxing occurs mean that if you call `GetType()` on a value type value, it always needs to be boxed first. Normally, that's a little inefficient but doesn't cause any confusion. With nullable value types, it'll either cause a `NullReferenceException` or return the underlying non-nullable value type. The following listing shows examples of these.

Listing 2.10 Calling `GetType` on nullable values leads to surprising results

```
Nullable<int> noValue = new Nullable<int>();
// Console.WriteLine(noValue.GetType());                                ← Would throw
Nullable<int> someValue = new Nullable<int>(5);
Console.WriteLine(someValue.GetType());                                     ← Prints System.Int32, the same
                                                                           as if you'd used typeof(int)
```

You've seen framework support and CLR support, but the C# language goes even further to make nullable value types easier to work with.

2.2.3 Language support

It would've been possible for C# 2 to have shipped with the compiler knowing only about nullable value types when enforcing the `struct` type constraint. It would've been awful, but it's useful to consider the absolute minimum support required in order to appreciate all the features that have been added to make nullable value types fit into the language more idiomatically. Let's start with the simplest part: simplifying nullable value type names.

THE ? TYPE SUFFIX

If you add a `?` to the end of the name of a non-nullable value type, that's precisely equivalent to using `Nullable<T>` for the same type. It works for the keyword shortcuts for the simple types (`int`, `double`, and so forth) as well as full type names. For example, these four declarations are precisely equivalent:

- `Nullable<int> x;`
- `Nullable<Int32> x;`
- `int? x;`
- `Int32? x;`

You can mix and match them however you like. The generated IL won't change at all. In practice, I end up using the `?` suffix everywhere, but other teams may have different conventions. For clarity, I've used `Nullable<T>` within the remainder of the text here, because the `?` can become confusing when used in prose, but in code that's rarely an issue.

That's the simplest language enhancement, but the theme of allowing you to write concise code continues through the rest of this section. The `?` suffix is about expressing a type easily; the next feature focuses on expressing a value easily.

THE NULL LITERAL

In C# 1, the expression `null` always referred to a null reference. In C# 2, that meaning is expanded to a null value: either a null reference or a value of a nullable value type where `HasValue` is `false`. This can be used for assignments, method arguments, comparisons—any manner of places. It's important to understand that when it's used for a nullable value type, it really does represent the value of that type where `HasValue` is `false` rather than being a null reference; if you try to work null references into your mental model of nullable value types, it'll get confusing quickly. The following two lines are equivalent:

```
int? x = new int?();  
  
int? x = null;
```

I typically prefer to use the null literal over explicitly calling the parameterless constructor (I'd write the second of the preceding lines rather than the first), but when it comes to comparisons, I'm ambivalent about the two options. For example, these two lines are equivalent:

```
if (x != null)  
  
if (x.HasValue)
```

I suspect I'm not even consistent about which I use. I'm not advocating for inconsistency, but this is an area where it doesn't hurt very much. You can always change your mind later with no compatibility concerns.

CONVERSIONS

You've already seen that `Nullable<T>` provides an implicit conversion from `T` to `Nullable<T>` and an explicit conversion from `Nullable<T>` to `T`. The language takes that set of conversions further by allowing certain conversions to chain together. Where there are two non-nullable value types `S` and `T` and there's a conversion from `S` to `T` (for example, the conversion from `int` to `decimal`), the following conversions are also available:

- `Nullable<S>` to `Nullable<T>` (implicit or explicit, depending on the original conversion)
- `S` to `Nullable<T>` (implicit or explicit, depending on the original conversion)
- `Nullable<S>` to `T` (always explicit)

These work in a reasonably obvious way by propagating null values and using the `S` to `T` conversion as required. This process of extending an operation to propagate nulls appropriately is called *lifting*.

One point to note: it's possible to explicitly provide conversions to both nullable and non-nullable types. LINQ to XML uses this to great effect. For example, there are explicit conversions from `XElement` to both `int` and `Nullable<int>`. Many operations in LINQ to XML will return a null reference if you ask them to find an element

that doesn't exist, and the conversion to `Nullable<int>` converts a null reference to a null value and propagates the nullity without throwing an exception. If you try to convert a null `XElement` reference to the non-nullable `int` type, however, an exception will be thrown. The existence of both conversions makes it easy to handle optional and required elements safely.

Conversions are one form of operator that can be built into C# or user-defined. Other operators defined on non-nullable types receive a similar sort of treatment in their nullable counterparts.

LIFTED OPERATORS

C# allows the following operators to be overloaded:

- Unary: `++` `--` `!` `~` `true` `false`
- Binary:⁵ `+` `-` `*` `/` `%` `&` `|` `^` `<<` `>>`
- Equality: `==` `!=`
- Relational: `<` `>` `<=` `>=`

When these operators are overloaded for a non-nullable value type `T`, the `Nullable<T>` type has the same operators with slightly different operand and result types. These are called *lifted operators* whether they're predefined operators, such as addition on numeric types, or user-defined operators, such as adding a `TimeSpan` to a `DateTime`. A few restrictions apply:

- The `true` and `false` operators are never lifted. They're incredibly rare in the first place, though, so this is no great loss.
- Only operators with non-nullable value types for the operands are lifted.
- For the unary and binary operators (other than equality and relational operators), the return type of the original operator has to be a non-nullable value type.
- For the equality and relational operators, the return type of the original operator has to be `bool`.
- The `&` and `|` operators on `Nullable<bool>` have separately defined behaviors, which we'll consider presently.

For all the operators, the operand types become their nullable equivalents. For the unary and binary operators, the return type also becomes nullable, and a null value is returned if any of the operands is a null value. The equality and relational operators keep their non-nullable Boolean return types. For equality, two null values are considered equal, and a null value and any non-null value are considered different. The relational operators always return `false` if either operand is a null value. When neither of the operands is a null value, the operator of the non-nullable type is invoked in the obvious way.

⁵ The equality and relational operators are also binary operators, but they behave slightly differently from the others, hence their separation in this list.

All these rules sound more complicated than they are; for the most part, everything works as you probably expect it to. It's easiest to see what happens with a few examples, and because `int` has so many predefined operators (and integers can be so easily expressed), it's the natural demonstration type. Table 2.1 shows a number of expressions, the lifted operator signature, and the result. It's assumed that there are variables `four`, `five`, and `nullInt`, each with type `Nullable<int>` and with the obvious values.

Table 2.1 Examples of lifted operators applied to nullable integers

Expression	Lifted operator	Result
<code>-nullInt</code>	<code>int? -(int? x)</code>	<code>null</code>
<code>-five</code>	<code>int? -(int? x)</code>	<code>-5</code>
<code>five + nullInt</code>	<code>int? +(int? x, int? y)</code>	<code>null</code>
<code>five + five</code>	<code>int? +(int? x, int? y)</code>	<code>10</code>
<code>four & nullInt</code>	<code>int? &(int? x, int? y)</code>	<code>null</code>
<code>four & five</code>	<code>int? &(int? x, int? y)</code>	<code>4</code>
<code>nullInt == nullInt</code>	<code>bool ==(int? x, int? y)</code>	<code>true</code>
<code>five == five</code>	<code>bool ==(int? x, int? y)</code>	<code>true</code>
<code>five == nullInt</code>	<code>bool ==(int? x, int? y)</code>	<code>false</code>
<code>five == four</code>	<code>bool ==(int? x, int? y)</code>	<code>false</code>
<code>four < five</code>	<code>bool <(int? x, int? y)</code>	<code>true</code>
<code>nullInt < five</code>	<code>bool <(int? x, int? y)</code>	<code>false</code>
<code>five < nullInt</code>	<code>bool <(int? x, int? y)</code>	<code>false</code>
<code>nullInt < nullInt</code>	<code>bool <(int? x, int? y)</code>	<code>false</code>
<code>nullInt <= nullInt</code>	<code>bool <=(int? x, int? y)</code>	<code>false</code>

Possibly the most surprising line of the table is the last one: that a null value isn't deemed less than or equal to another null value even though they are deemed to be equal to each other (as per the seventh row)! This is very odd, but it's unlikely to cause problems in real life, in my experience. In the list of restrictions regarding operator lifting, I mentioned that `Nullable<bool>` works slightly differently from the other types.

NULLABLE LOGIC

Truth tables are often used to demonstrate Boolean logic with all possible input combinations and the result. Although the same approach can be used for `Nullable<Boolean>` logic, we have three values to consider (`true`, `false`, and `null`) for each input instead of just `true` and `false`. There are no conditional logical operators (the short-circuiting `&&` and `||` operators) defined for `Nullable<bool>`, which makes life simpler.

Only the logical AND and inclusive OR operators (`&` and `|`, respectively) have special behavior. The other operators—unary logical negation (`!`) and exclusive OR

(\wedge)—follow the same rules as other lifted operators. For the sake of completeness, table 2.2 gives the truth table for all four valid `Nullable<bool>` logical operators. I've highlighted the results that would be different if the extra rules didn't exist for `Nullable<bool>`.

Table 2.2 Truth table for `Nullable<bool>` operators

x	y	<code>x & y</code>	<code>x y</code>	<code>x ^ y</code>	<code>!x</code>
true	true	true	true	false	false
true	false	false	true	true	false
true	null	null	true	null	false
false	true	false	true	true	true
false	false	false	false	false	true
false	null	false	null	null	true
null	true	null	true	null	null
null	false	false	null	null	null
null	null	null	null	null	null

If you find reasoning about rules easier to understand than looking up values in tables, the idea is that a null `bool?` value is in some senses a maybe. If you imagine that each null entry in the input side of the table is a variable instead, you'll always get a null value on the output side of the table if the result depends on the value of that variable. For instance, looking at the third line of the table, the expression `true & y` will be true only if `y` is true, but the expression `true | y` will always be true whatever the value of `y` is, so the nullable results are null and true, respectively.

When considering the lifted operators and particularly how nullable logic works, the language designers had two slightly contradictory sets of existing behavior: C# 1 null references and SQL NULL values. In many cases, these don't conflict at all; C# 1 had no concept of applying logical operators to null references, so there was no problem in using the SQL-like results given earlier. The definitions you've seen may surprise some SQL developers, though, when it comes to comparisons. In standard SQL, the result of comparing two values (in terms of equality or greater than/less than) is always unknown if either value is NULL. The result in C# 2 is never null, and two null values are considered to be equal to each other.

Results of lifted operators are specific to C#

The lifted operators and conversions, along with the `Nullable<bool>` logic described in this section, are all provided by the C# compiler and not by the CLR or the framework itself. If you use ildasm on code that evaluates any of these nullable operators, you'll find that the compiler has created all the appropriate IL to test for null values and dealt with them accordingly.

(continued)

Different languages can behave differently on these matters, and this is definitely something to look out for if you need to port code between different .NET-based languages. For example, VB treats lifted operators far more like SQL, so the result of `x < y` is `Nothing` if `x` or `y` is `Nothing`.

Another familiar operator is now available with nullable value types, and it behaves as you'd probably expect it to if you consider your existing knowledge of null references and just tweak it to be in terms of null values.

THE AS OPERATOR AND NULLABLE VALUE TYPES

Prior to C# 2, the `as` operator was available only for reference types. As of C# 2, it can now be applied to nullable value types as well. The result is a value of that nullable type: the `null` value if the original reference was the wrong type or null or a meaningful value otherwise. Here's a short example:

```
static void PrintValueAsInt32(object o)
{
    int? nullable = o as int?;
    Console.WriteLine(nullable.HasValue ?
        nullable.Value.ToString() : "null");
}
...
PrintValueAsInt32(5);           ↪ Prints 5
PrintValueAsInt32("some string"); ↪ Prints null
```

This allows you to safely convert from an arbitrary reference to a value in a single step, although you'd normally check whether the result is `null` afterward. In C# 1, you'd have had to use the `is` operator followed by a cast, which is inelegant; it's essentially asking the CLR to perform the same type check twice.

NOTE Using the `as` operator with nullable types is surprisingly slow. In most code, this is unlikely to matter (it's not going to be slow compared with any I/O, for example), but it's slower than `is` and then a cast in all the framework and compiler combinations I've tried.

C# 7 has an even better solution for most cases where I've used the `as` operator with nullable value types using pattern matching (described in chapter 12). If your intended result type really is a `Nullable<T>`, though, the `as` operator is handy. Finally, C# 2 introduced an entirely new operator specifically for handling null values elegantly.

THE NULL-COALEScing ?? OPERATOR

It's reasonably common to want to use nullable value types—or indeed, reference types—and provide a sort of default value if a particular expression evaluates to `null`. C# 2 introduced the `??` operator, also known as the *null-coalescing operator*, for precisely this purpose.

`??` is a binary operator that evaluates an expression of `first ?? second` by going through the following steps (roughly speaking):

- 1 Evaluate `first`.
- 2 If the result is non-null, that's the result of the whole expression.
- 3 Otherwise, evaluate `second`, and use that as the result of the whole expression.

I say roughly speaking because the formal rules in the specification have to deal with situations involving conversions between the types of `first` and `second`. These aren't important in most uses of the operator, and I don't intend to go through them. They're easy to find in the specification if you need them.

One aspect of those rules is worth highlighting. If the type of the first operand is a nullable value type and the type of the second operand is the underlying type of the first operand, the type of the whole expression is that (non-nullable) underlying type. For example, this code is perfectly valid:

```
int? a = 5;
int b = 10;
int c = a ?? b;
```

Note that you're assigning directly to `c` even though its type is the non-nullable `int` type. You can do this only because `b` is non-nullable, so you know that the overall result can't be null. The `??` operator composes well with itself; an expression such as `x ?? y ?? z` will evaluate `y` only if `x` evaluates to null and will evaluate `z` only if both `x` and `y` evaluate to null.

Null values become even easier to work with—and more likely as expression results—in C# 6 with the `?.` null conditional operator, as you'll see in section 10.3. Combining `?.` and `??` can be a powerful way of handling possible nulls at various points of execution. Like all techniques, this is best used in moderation. If you find your code's readability going downhill, you might want to consider using multiple statements to avoid trying to do too much in one go.

That's it for nullable value types in C# 2. We've now covered the two most important features of C# 2, but we have a couple of fairly large features still to talk about, along with a raft of smaller ones. Next up is delegates.

2.3 Simplified delegate creation

The basic purpose of delegates hasn't changed since they were first introduced: to encapsulate a piece of code so that it can be passed around and executed as necessary in a type-safe fashion in terms of the return type and parameters. Back in the days of C# 1, that was almost always used for event handling or starting threads. This was mostly still the case when C# 2 was introduced in 2005. It was only in 2008 that LINQ helped C# developers feel comfortable with the idea of passing a function around for all kinds of reasons.

C# 2 brought three new ways of creating delegate instances as well as the ability to declare generic delegates, such as `EventHandler<TEventArgs>` and `Action<T>`. We'll start with method group conversions.

2.3.1 Method group conversions

A *method group* refers to one or more methods with the same name. Every C# developer has been using them forever without necessarily thinking about it, because every method invocation uses one. For example, consider this trivial code:

```
Console.WriteLine("hello");
```

The expression `Console.WriteLine` is a method group. The compiler then looks at the arguments to work out which of the overloads within that method group should be invoked. Other than method invocations, C# 1 used method groups in *delegate creation expressions* as the only way the language provided to create a delegate instance. For example, say you have a method like this:

```
private void HandleButtonClick(object sender, EventArgs e)
```

Then you could create an `EventHandler`⁶ instance like this:

```
EventHandler handler = new EventHandler(HandleButtonClick);
```

C# 2 introduced *method group conversions* as a sort of shorthand: a method group is implicitly convertible to any delegate type with a signature that's compatible with one of the overloads. You'll explore the notion of compatibility further in section 2.3.3, but for the moment you'll look at methods that exactly match the signature of the delegate you're trying to convert to.

In the case of our preceding `EventHandler` code, C# 2 allows you to simplify the creation of the delegate to this:

```
EventHandler handler = HandleButtonClick;
```

This works for event subscription and removal, too:

```
button.Click += HandleButtonClick;
```

The same code is generated as for the delegate creation expression, but it's much more concise. These days, I rarely see delegate creation expressions in idiomatic code. Method group conversions save a few characters when creating a delegate instance, but anonymous methods achieve a lot more.

2.3.2 Anonymous methods

You might reasonably expect a lot of detail on anonymous methods here. I'm going to save most of that information for the successor of anonymous methods: lambda expressions. They were introduced in C# 3, and I expect that if they'd existed before anonymous methods, the latter would never have been introduced at all.

⁶ For reference, `EventHandler` has a signature of `public delegate void EventHandler(object sender, EventArgs e)`.

Even so, their introduction in C# 2 made me think about delegates in a whole different way. Anonymous methods allow you to create a delegate instance without having a real method to refer to⁷ just by writing some code inline wherever you want to create the instance. You just use the `delegate` keyword, optionally include some parameters, and then write some code in braces. For example, if you wanted an event handler that just logged to the console when it was fired, you could do that very simply:

```
EventHandler handler = delegate
{
    Console.WriteLine("Event raised");
};
```

That doesn't call `Console.WriteLine` immediately; instead it creates a delegate that'll call `Console.WriteLine` when it's invoked. To see the type of the sender and event arguments, you need appropriate parameters:

```
EventHandler handler = delegate(object sender, EventArgs args)
{
    Console.WriteLine("Event raised. sender={0}; args={1}",
        sender.GetType(), args.GetType());
};
```

The real power comes when you use an anonymous method as a *closure*. A closure is able to access all the variables that are in scope at the point of its declaration, even if those variables normally wouldn't be available anymore when the delegate is executed. You'll look at closures in a lot more detail (including how the compiler treats them) when you look at lambda expressions. For now, here's a single brief example; it's an `AddClickLogger` method that adds a `Click` handler to any control with a custom message that's passed into `AddClickLogger`:

```
void AddClickLogger(Control control, string message)
{
    control.Click += delegate
    {
        Console.WriteLine("Control clicked: {0}", message);
    }
}
```

Here the `message` variable is a parameter to the method, but it's captured by the anonymous method. The `AddClickLogger` method doesn't execute the event handler itself; it just adds it as a handler for the `Click` event. By the time the code in the anonymous method executes, `AddClickLogger` will have returned. How does the parameter still exist? In short, the compiler handles it all for you to avoid you having to write boring code. Section 3.5.2 provides more details when you look at capturing variables in lambda expressions. There's nothing special about `EventHandler` here; it's just a well-known delegate type that's been part of the framework forever. For the final part

⁷ In your source code, anyway. The method still exists in the IL.

of our whirlwind tour of C# 2 delegate improvements, let's come back to the idea of compatibility, which I mentioned when talking about method group conversions.

2.3.3 Delegate compatibility

In C# 1, you needed a method with a signature with exactly the same return type and parameter types (and `ref`/`out` modifiers) to create a delegate instance. For example, suppose you had this delegate declaration and method:

```
public delegate void Printer(string message);

public void PrintAnything(object obj)
{
    Console.WriteLine(obj);
}
```

Now imagine you wanted to create an instance of `Printer` to effectively wrap the `PrintAnything` method. It feels like it should be okay; a `Printer` will always be given a `string` reference, and that's convertible to an `object` reference via an identity conversion. C# 1 wouldn't allow that, though, because the parameter types don't match. C# 2 allows this for delegate creation expressions and for method group conversions:

```
Printer p1 = new Printer(PrintAnything);
Printer p2 = PrintAnything;
```

Additionally, you can create one delegate to wrap another one with a compatible signature. Suppose you had a second delegate type that coincidentally did match the `PrintAnything` method:

```
public delegate void GeneralPrinter(object obj);
```

If you already have a `GeneralPrinter`, you can create a `Printer` from it:

```
GeneralPrinter generalPrinter = ...;
Printer printer = new Printer(generalPrinter);
```

The compiler lets you do that because it's safe; any argument that can be passed to a `Printer` can safely be passed to a `GeneralPrinter`. The compiler is happy to do the same in the other direction for return types, as shown in the following example:

```
public delegate object ObjectProvider();
public delegate string StringProvider();

StringProvider stringProvider = ...;
ObjectProvider objectProvider =
    new ObjectProvider(stringProvider);
```

Parameterless delegates returning values

Again, this is safe because any value that `StringProvider` can return would definitely be fine to return from an `ObjectProvider`.

It doesn't always work the way you might want it to, though. The compatibility between different parameter or return types has to be in terms of an *identity conversion* that doesn't change the representation of the value at execution time. For example, this code doesn't compile:

```
public delegate void Int32Printer(int x);
public delegate void Int64Printer(long x);
```

Delegates accepting 32-
and 64-bit integers

```
Int64Printer int64Printer = ...;
Int32Printer int32Printer =
    new Int32Printer(int64Printer);
```

Any way you might
create an `Int64Printer`

Error! Can't wrap the
`Int64Printer` in an `Int32Printer`

The two delegate signatures here aren't compatible; although there's an implicit conversion from `int` to `long`, it's not an identity conversion. You might argue that the compiler could've silently created a method that performed the conversion for you, but it doesn't do so. In a way, that's helpful, because this behavior fits in with the *generic variance* feature you'll see in chapter 4.

It's important to understand that although this feature looks a bit like generic variance, they are different features. Aside from anything else, this wrapping really does create a new instance of the delegate instead of just treating the existing delegate as an instance of a different type. I'll go into more detail when you look at the feature fully, but I wanted to highlight as early as possible that they're not the same.

That's it for delegates in C# 2. Method group conversions are still widely used, and often the compatibility aspect will be used without anyone even thinking about it. Anonymous methods aren't seen much these days, because lambda expressions can do almost anything anonymous methods can, but I still look on them fondly as my first taste of the power of closures. Speaking of one feature that led to another, let's look at the forerunner of C# 5's asynchrony: iterator blocks.

2.4 Iterators

Relatively few interfaces have specific language support in C# 2. `IDisposable` has support via the `using` statement, and the language makes guarantees about the interfaces that arrays implement, but apart from that, only the enumerable interfaces have direct support. `IEnumerable` has always had support for consumption in the form of the `foreach` statement, and C# 2 extended that to its new-to-.NET-2 generic counterpart `IEnumerable<T>` in a reasonably obvious way.

The enumerable interfaces represent sequences of items, and although consuming them is extremely common, it's also entirely reasonable to want to *produce* a sequence. Implementing either the generic or nongeneric interfaces manually can be tedious and error prone, so C# 2 introduced a new feature called *iterators* to make it simpler.

2.4.1 Introduction to iterators

An *iterator* is a method or property implemented with an *iterator block*, which is in turn just a block of code using the `yield return` or `yield break` statements. Iterator blocks can be used only to implement methods or properties with one of the following return types:

- `IEnumerable`
- `IEnumerable<T>` (where `T` can be a type parameter or a regular type)
- `IEnumerator`
- `IEnumerator<T>` (where `T` can be a type parameter or a regular type)

Each iterator has a *yield type* based on its return type. If the return type is one of the non-generic interfaces, the yield type is `object`. Otherwise, it's the type argument provided to the interface. For example, the yield type of a method returning `IEnumerable<string>` is `string`.

The `yield return` statements provide values for the returned sequence, and a `yield break` statement will terminate a sequence. Similar constructs, sometimes called *generators*, exist in some other languages, such as Python.

The following listing shows a simple iterator method that you can analyze further. I've highlighted the `yield return` statements in the method.

Listing 2.11 A simple iterator yielding integers

```
static IEnumerable<int> CreateSimpleIterator()
{
    yield return 10;
    for (int i = 0; i < 3; i++)
    {
        yield return i;
    }
    yield return 20;
}
```

With that method in place, you can call the method and iterate over the results with a regular `foreach` loop:

```
foreach (int value in CreateSimpleIterator())
{
    Console.WriteLine(value);
}
```

That loop will print the following output:

```
10
0
1
2
20
```

So far, this isn't terribly exciting. You could change the method to create a `List<int>`, replace each `yield return` statement with a call to `Add()`, and then return the list at the end of the method. The loop output would be exactly the same, but it wouldn't execute in the same way at all. The huge difference is that iterators are executed lazily.

2.4.2 Lazy execution

Lazy execution, or lazy evaluation, was invented as part of lambda calculus in the 1930s. The basic idea of it is simple: execute code only when you need the value that it'll compute. There are uses of it well beyond iterators, but they're all we need it for right now.

To explain how the code executes, the following listing expands the `foreach` loop into mostly equivalent code that uses a `while` loop instead. I've still used the syntactic sugar of a `using` statement that will call `Dispose` automatically, just for simplicity.

Listing 2.12 The expansion of a foreach loop

```
IEnumerable<int> enumerable = CreateSimpleIterator();
using (IEnumerator<int> enumerator =
    enumerable.GetEnumerator())
{
    while (enumerator.MoveNext())
    {
        int value = enumerator.Current;
        Console.WriteLine(value);
    }
}
```

If you've never looked at the `IEnumerable/IEnumerator` pair of interfaces (and their generic equivalents) before, now is a good time to make sure you understand the difference between them. An `IEnumerable` is a sequence that can be iterated over, whereas an `IEnumerator` is like a cursor within a sequence. Multiple `IEnumerator` instances can probably iterate over the same `IEnumerable` without changing its state at all. Compare that with an `IEnumerator`, which naturally *does* have mutable state: each time you call `MoveNext()`, you're asking it to move the cursor to the next element of the sequence it's iterating over.

If that didn't make much sense, you might want to think about an `IEnumerable` as a book and an `IEnumerator` as a bookmark. There can be multiple bookmarks within a book at any one time. Moving a bookmark to the next page doesn't change the book or any of the other bookmarks, but it does change that bookmark's state: its position within the book. The `IEnumerable.GetEnumerator()` method is a sort of bootstrapping: it asks the sequence to create an `IEnumerator` that's set up to iterate over that sequence, just like putting a new bookmark at the start of a book.

After you have an `IEnumerator`, you repeatedly call `MoveNext()`; if it returns `true`, that means you've moved to another value that you can access with the `Current` property. If `MoveNext()` returns `false`, you've reached the end of the sequence.

What does this have to do with lazy evaluation? Well, now that you know exactly what the code using the iterator will call, you can look at when the method body starts executing. Just as a reminder, here's the method from listing 2.11:

```
static IEnumerable<int> CreateSimpleIterator()
{
    yield return 10;
    for (int i = 0; i < 3; i++)
    {
        yield return i;
    }
    yield return 20;
}
```

When `CreateSimpleIterator()` is called, none of the method body is executed.

If you put a breakpoint on the first line (`yield return 10`) and step through the code, you won't hit the breakpoint when you call the method. You won't hit the breakpoint when you call `GetEnumerator()`, either. The method body starts executing only when `MoveNext()` is called. But what happens then?

2.4.3 Evaluation of yield statements

Even when the method starts executing, it goes only as far as it needs to. It stops executing when any of the following occurs:

- An exception is thrown.
- It reaches the end of the method.
- It reaches a `yield break` statement.
- It has evaluated the operand to a `yield return` statement, so it is ready to yield the value.

If an exception is thrown, that exception is propagated as normal. If the end of the method is reached or it hits a `yield break` statement, the `MoveNext()` method returns `false` to indicate that you've reached the end of the sequence. If you reach a `yield return` statement, the `Current` property is set to the value you're yielding, and `MoveNext()` returns `true`.

NOTE To clarify the preceding paragraph, the exception is propagated as normal, assuming you're already executing the iterator code. Don't forget that until the calling code iterates over the returned sequence, you won't start executing the iterator code. It's the `MoveNext()` call that will throw the exception, not the initial call to the iterator method.

In our simple example, as soon as `MoveNext()` starts iterating, it reaches the `yield return 10;` statement, sets `Current` to 10, and then returns `true`.

That all sounds simple for the first call to `MoveNext()`, but what about subsequent ones? You can't start again from scratch; otherwise, the sequence would be 10 repeated an infinite number of times. Instead, when `MoveNext()` returns, it's as if the method is paused. The generated code keeps track of the point you've reached in the method along with any other state, such as the local variable `i` in your loop. When `MoveNext()` is called again, execution picks up from the point you've reached and keeps going. That's what makes it lazy, and that's the part that's difficult to get right when you're writing the code yourself.

2.4.4 The importance of being lazy

To give you an idea of why this is important, let's write some code to print out the Fibonacci sequence until you hit the first value over 1,000. The following listing shows a `Fibonacci()` method that returns an infinite sequence and then a method that iterates over that sequence until it hits a limit.

Listing 2.13 Iterating over the Fibonacci sequence

```
static IEnumerable<int> Fibonacci()
{
    int current = 0;
    int next = 1;
    while (true)
    {
        yield return current;
        int oldCurrent = current;
        current = next;
        next = next + oldCurrent;
    }
}

static void Main()
{
    foreach (var value in Fibonacci())
    {
        Console.WriteLine(value);
        if (value > 1000)
        {
            break;
        }
    }
}
```

How would you do something like this without iterators? You could change the method to create a `List<int>` and populate it until you hit the limit. But that list could be big if the limit is large, and why should the method that knows the details of the Fibonacci sequence also know how you want to stop? Suppose you sometimes want to stop based on how long you've been printing out values, sometimes based on how many values you've printed, and sometimes based on the current value. You don't want to implement the method three times.

You could avoid creating the list by printing the value in the loop, but that makes your `Fibonacci()` method even more tightly coupled to the one thing you happen to want to do with the values right now. What if you wanted to add the values together instead of printing them? Would you write a second method? It's all a ghastly violation of the separation of concerns.

The iterator solution is exactly what you want: a representation of an infinite sequence, and that's all. The calling code can iterate over it as far as it wants⁸ and use the values however it wants.

Implementing the Fibonacci sequence manually wouldn't be terribly hard. There's little state to maintain between calls, and the flow control is simple. (The fact that there's only one `yield return` statement helps there.) But as soon as the code gets more complicated, you don't want to be writing this code yourself. The compiler not only generates code that keeps track of where the code has reached, but it's also smart about how to handle `finally` blocks, which aren't quite as simple as you might think.

2.4.5 Evaluation of finally blocks

It may seem odd that I'd focus on `finally` blocks out of all the syntax that C# has for managing execution flow, but the way that they're handled in iterators is both interesting and important for the usefulness of the feature. In reality, it's far more likely that you'll use `using` statements than the raw `finally` blocks, but you can view `using` statements as effectively built with `finally` blocks, so the same behavior holds.

To demonstrate how the execution flow works, the following listing shows a trivial iterator block that yields two items within a `try` block and writes its progress to the console. You'll then use the method in a couple of ways.

Listing 2.14 An iterator that logs its progress

```
static IEnumerable<string> Iterator()
{
    try
    {
        Console.WriteLine("Before first yield");
        yield return "first";
        Console.WriteLine("Between yields");
        yield return "second";
        Console.WriteLine("After second yield");
    }
    finally
    {
        Console.WriteLine("In finally block");
    }
}
```

⁸ At least until it overflows the range of `int`. At that point, it might throw an exception or underflow to a large negative number depending on whether the code is in a checked context.

Before you run it, think about what you'd expect this to print if you just iterate over the sequence returned by the method. In particular, would you expect to see `In finally block` in the console when `first` is returned? There are two ways of thinking about it:

- If you consider execution to be paused by the `yield return` statement, then logically it's still inside the `try` block, and there's no need to execute the `finally` block.
- If you think about the code having to actually return to the `MoveNext()` caller when it hits the `yield return` statement, then it feels like you're exiting the `try` block and should execute the `finally` block as normal.

Without wanting to spoil the surprise, the pause model wins. It's much more useful and avoids other aspects that seem counterintuitive. It would be odd to execute each statement in a `try` block just once but execute its `finally` block three times, for example—once for each time you yield a value and then when you execute the rest of the method.

Let's prove that it works that way. The following listing calls the method and iterates over the values in the sequence and prints them as it goes.

Listing 2.15 A simple `foreach` loop to iterate and log

```
static void Main()
{
    foreach (string value in Iterator())
    {
        Console.WriteLine("Received value: {0}", value);
    }
}
```

The output of listing 2.15 shows that the `finally` block is executed only once at the end:

```
Before first yield
Received value: first
Between yields
Received value: second
After second yield
In finally block
```

This also proves that lazy evaluation is working: the output from the `Main()` method is interleaved with the output from the `Iterator()` method, because the iterator is repeatedly paused and resumed.

So far, so simple, but that relied on you iterating through the whole of the sequence. What if you want to stop halfway through? If the code that's fetching items from an iterator calls `MoveNext()` only once (if it needs only the first value from the sequence, for example), does that leave the iterator paused in the `try` block forever without ever executing the `finally` block?

The answer is yes and no. If you write all the calls to the `IEnumerator<T>` manually and call `MoveNext()` just once, the finally block will indeed never get executed. But if you write a `foreach` loop and happen to exit it without looping over the whole sequence, the `finally` block *will* get executed. The following listing demonstrates that by breaking out of the loop as soon as it sees a non-null value (which it will do immediately, of course). It's the same as listing 2.15 but with the addition of the part in bold.

Listing 2.16 Breaking out of a `foreach` loop by using an iterator

```
static void Main()
{
    foreach (string value in Iterator())
    {
        Console.WriteLine("Received value: {0}", value);
        if (value != null)
        {
            break;
        }
    }
}
```

The output of listing 2.16 is as follows:

```
Before first yield
Received value: first
In finally block
```

The last line is the important one: you're still executing the `finally` block. That happens automatically when you exit the `foreach` loop, because that has a hidden `using` statement. Listing 2.17 shows what listing 2.16 would look like if you couldn't use a `foreach` loop and had to write the equivalent code by hand. If this looks familiar, it's because you did the same thing in listing 2.12, but this time you're paying more attention to the `using` statement.

Listing 2.17 Expansion of listing 2.16 to not use a `foreach` loop

```
static void Main()
{
    IEnumerable<string> enumerable = Iterator();
    using (IEnumerator<string> enumerator = enumerable.GetEnumerator())
    {
        while (enumerator.MoveNext())
        {
            string value = enumerator.Current;
            Console.WriteLine("Received value: {0}", value);
            if (value != null)
            {
                break;
            }
        }
    }
}
```

The important part is the `using` statement. That makes sure that however you leave it, you'll call `Dispose` on the `IEnumerator<string>`. If the iterator method is "paused" within the `try` block at that point, the `Dispose` method ends up executing the `finally` block. Isn't it clever?

2.4.6 *The importance of finally handling*

This may sound like a minor detail, but it makes a huge difference in how applicable iterators are. It means they can be used for methods that acquire resources that need disposing of, such as file handles. It also means that they can be used to chain to other iterators with the same requirement. You'll see in chapter 3 that LINQ to Objects uses sequences a lot, and reliable disposal is crucial to being able to work with files and other resources.

All of this requires the caller to dispose of the iterator

If you don't call `Dispose` on an iterator (and you haven't iterated to the end of the sequence), you can leak resources or at least delay cleanup. This should be avoided.

The nongeneric `IEnumerator` interface doesn't extend `IDisposable`, but the `foreach` loop checks whether the runtime implementation also implements `IDisposable`, and calls `Dispose` if necessary. The generic `IEnumerator<T>` interface does extend `IDisposable`, making things simpler.

If you're iterating by calling `MoveNext()` manually (which can definitely have its place), you should do the same thing. If you're iterating over a generic `IEnumerable<T>`, you can just use a `using` statement as I have in my expanded `foreach` loop listings. If you're in the unfortunate position of iterating over a nongeneric sequence, you should perform the same interface check that the compiler does in `foreach`.

As an example of how useful it can be to acquire resources in iterator blocks, consider the following listing of a method that returns a sequence of lines read from a file.

Listing 2.18 Reading lines from a file

```
static IEnumerable<string> ReadLines(string path)
{
    using (TextReader reader = File.OpenText(path))
    {
        string line;
        while ((line = reader.ReadLine()) != null)
        {
            yield return line;
        }
    }
}
```

A method like this was introduced in .NET 4.0 (`File.ReadLines`), but the framework method doesn't work well if you call the method once but iterate over the result

multiple times; it opens the file only once. The method in listing 2.18 opens the file each time you iterate, making it simpler to reason about. This has the downside, however, of delaying any exception due to the file not existing or not being readable. Tricky trade-offs always exist in API design.

The point of showing you this method is to demonstrate how important it is that iterator disposal is handled properly. If a `foreach` loop that threw an exception or returned early resulted in a dangling open file handle, the method would be close to useless. Before we leave iterators, let's peek behind the curtain briefly and see how they're implemented.

2.4.7 Implementation sketch

I always find it useful to see roughly what the compiler does with code, particularly for complicated situations such as iterators, `async/await`, and anonymous functions. This section provides only a taste; an article at <http://csharpindepth.com> provides far more detail. Please be aware that the exact details are implementation specific; you may find different compilers take slightly different approaches. I'd expect most to have the same basic strategy, though.

The first thing to understand is that even though you've written a method,⁹ the compiler generates a whole new type for you to implement the relevant interfaces. Your method body is moved into a `MoveNext()` method in this generated type and adjusted for the execution semantics of iterators. To demonstrate the generated code, we'll look at the code that the compiler generates for the following listing.

Listing 2.19 Sample iterator method to decompile

```
public static IEnumerable<int> GenerateIntegers(int count)
{
    try
    {
        for (int i = 0; i < count; i++)
        {
            Console.WriteLine("Yielding {0}", i);
            yield return i;
            int doubled = i * 2;
            Console.WriteLine("Yielding {0}", doubled);
            yield return doubled;
        }
    }
    finally
    {
        Console.WriteLine("In finally block");
    }
}
```

⁹ You can use iterators to write property accessors as well, but I'll just talk about iterator methods for the rest of this section, just to be concise. The implementation is the same for property accessors.

Listing 2.19 shows a relatively simple method in its original form, but I've deliberately included five aspects that may not seem obvious:

- A parameter
- A local variable that needs to be preserved across `yield return` statements
- A local variable that doesn't need to be preserved across `yield return` statements
- Two `yield return` statements
- A `finally` block

The method iterates over its loop count times and yields two integers on each iteration: the iteration number and double the same value. For example, if you pass in 5, it will yield 0, 0, 1, 2, 2, 4, 3, 6, 4, 8.

The downloadable source code contains a full, manually tweaked, decompiled form of the generated code. It's pretty long, so I haven't included it in its entirety here. Instead, I want to give you a flavor of what's generated. The following listing shows most of the infrastructure but none of the implementation details. I'll explain that, and then you'll look at the `MoveNext()` method, which does most of the real work.

Listing 2.20 Infrastructure of the generated code for an iterator

```
public static IEnumerable<int> GenerateIntegers(
    int count)
{
    GeneratedClass ret = new GeneratedClass(-2);
    ret.count = count;
    return ret;
}

private class GeneratedClass
    : IEnumerable<int>, IEnumerator<int>
{
    public int count;
    private int state;
    private int current;
    private int initialThreadId;
    private int i;

    public GeneratedClass(int state)
    {
        this.state = state;
        initialThreadId = Environment.CurrentManagedThreadId;
    }

    public bool MoveNext() { ... }

    public IEnumerator<int> GetEnumerator() { ... }
    public void Reset()
    {
        throw new NotSupportedException();
    }
}
```

Stub method with the original declared signature

Generated class to represent the state machine

All the fields in the state machine with varying purposes

Constructor called by both the stub method and GetEnumerator

Main body of state machine code

Creates a new state machine if necessary

Generated iterators never support Reset

```

public void Dispose() { ... }           ← Executes any finally
                                         blocks, if required
public int Current { get { return current; } }   ← Current property to
                                                 return last-yielded value
private void Finally() { ... }           ← Body of a finally block for use
                                         in MoveNext and Dispose
IEnumerator Enumerable().GetEnumerator()
{
    return GetEnumerator();
}

object IEnumerator.Current { get { return current; } }
}

```

Explicit implementation of nongeneric interface members

Yes, that's the simplified version. The important point to understand is that the compiler generates a *state machine* for you, as a private nested class. A lot of the names generated by the compiler aren't valid C# identifiers, but I've provided valid ones for simplicity. The compiler still emits a method with the signature declared in the original source code, and that's what any callers will use. All that does is create an instance of the state machine, copy any parameter to it, and return the state machine to the caller. None of the original source code is called, which corresponds to the lazy behavior you've already seen.

The state machine contains everything it needs to implement the iterator:

- An indicator of where you are within the method. This is similar to an instruction counter in a CPU but simpler because you need to distinguish between only a few states
- A copy of all the parameters, so you can obtain their values when you need them
- Local variables within the method
- The last-yielded value, so the caller can obtain it with the `Current` property

You'd expect the caller to perform the following sequence of operations:

- 1 Call `GetEnumerator()` to obtain an `IEnumerator<int>`.
- 2 Repeatedly call `MoveNext()` and then `Current` on the `IEnumerator<int>`, until `MoveNext()` returns `false`.
- 3 Call `Dispose` for any cleanup that's required, whether an exception was thrown or not.

In almost all cases, the state machine is used only once and only on the same thread it was created on. The compiler generates code to optimize for this case; the `GetEnumerator()` method checks for it and returns `this` if the state machine is still in its original state and is on the same thread. That's why the state machine implements both `IEnumerable<int>` and `IEnumerator<int>`, which would be unusual to see in normal code.¹⁰ If `GetEnumerator()` is called from a different thread or multiple

¹⁰ If the original method returns only `IEnumerator<T>`, the state machine implements only that.

times, those calls create a new instance of the state machine with the initial parameter values copied in.

The `MoveNext()` method is the complicated bit. The first time it's called, it just needs to start executing the code written in the method as normal; but on subsequent calls, however, it needs to effectively jump to the right point in the method. The local variables need to be preserved between calls as well, so they're stored in fields in the state machine.

In an optimized build, some local variables don't have to be copied into fields. The point of using a field is so you can keep track of the value you set in one `MoveNext()` call when you come back in the next `MoveNext()` call. If you look at the doubled local variable from listing 2.19, it's never used like that:

```
for (int i = 0; i < count; i++)
{
    Console.WriteLine("Yielding {0}", i);
    yield return i;
    int doubled = i * 2;
    Console.WriteLine("Yielding {0}", doubled);
    yield return doubled;
}
```

All you do is initialize the variable, print it out, and then yield it. When you return to the method, that value is irrelevant so the compiler can optimize it into a real local variable in a release build. In a debug build, it may still be present to improve the debugging experience. Notice that if you swapped the last two bold lines in the preceding code—yielded the value and *then* printed it—the optimization wouldn't be possible.

What does a `MoveNext()` method look like? It's difficult to give real code without getting stuck in too much detail, so the following listing gives a sketch of the structure.

Listing 2.21 Simplified `MoveNext()` method

```
public bool MoveNext()
{
    try
    {
        switch (state)
        {
            // Jump table to get to
            // the right part of the
            // rest of the method
        }
    }
    fault
    {
        Dispose();
    }
}
```

Annotations for Listing 2.21:

- Jump table to get to the right part of the rest of the method**: Points to the `switch (state)` statement.
- Method code returning at each yield return**: Points to the brace after the `switch (state)` statement.
- Fault block executed only on exceptions**: Points to the `fault` block.
- Clean up on exceptions**: Points to the `Dispose();` call.

The state machine contains a variable (in our case, called `state`) that remembers where it reached. The precise values used depend on the implementation, but in the version of Roslyn I happened to use, the states were effectively as follows:

- `-3`—`MoveNext()` currently executing
- `-2`—`GetEnumerator()` not yet called
- `-1`—Completed (whether successfully or not)
- `0`—`GetEnumerator()` called but `MoveNext()` not yet called (start of method)
- `1`—At the first `yield return` statement
- `2`—At the second `yield return` statement

When `MoveNext()` is called, it uses this state to jump to the right place in the method to either start executing for the first time or resume from the previous `yield return` statement. Notice that there aren't any states for positions in the code such as "just assigned a value to the doubled variable," because you never need to resume from there; you need to resume only from where you previously paused.

The `fault` block near the end of listing 2.21 is an IL construct with no direct equivalent in C#. It's like a `finally` block that executes when an exception is thrown but without catching the exception. This is used to perform any cleanup operations required; in our case, that's the `finally` block. The code in that `finally` block is moved into a separate method that's called from `Dispose()` (if an exception has been thrown) and `MoveNext()` (if you reach it without an exception). The `Dispose()` method checks the state to see what cleanup is required. That becomes more complicated the more `finally` blocks there are.

Looking at the implementation isn't enlightening in terms of teaching you more C# coding techniques, but it's great for building an appreciation of how much the compiler is capable of doing on your behalf. The same ideas come into play again in C# 5 with `async/await`, where instead of pausing until the `MoveNext()` is called again, asynchronous methods effectively pause until an asynchronous operation has completed.

We've now covered the biggest features of C# 2, but several smaller features were introduced at the same time. These features are reasonably simple to describe, which is why I've lumped them all together here. They're not otherwise related, but sometimes that's just the way language design happens.

2.5 Minor features

Some of the features described in this section are rarely used in my experience, but others are common in any modern C# codebase. The time it takes to describe a feature doesn't always correlate with how useful it is. In this section, you'll look at the following:

- Partial types that allow code for a single type to be split across multiple source files
- Static classes for utility types
- Separate accessibility (public, private, and so on) for `get` and `set` accessors in properties

- Improvements to namespace aliases to make it easier to work with code that uses the same names in multiple namespaces or assemblies
- Pragma directives that allow additional compiler-specific features such as temporarily disabling warnings
- Fixed-size buffers for inline data in unsafe code
- The `[InternalsVisibleTo]` attribute, which makes testing simpler

Each feature is independent of the others, and the order in which I've described them is unimportant. If you know just enough about one of these sections to know it's irrelevant to you, you can safely skip it without that becoming a problem later.

2.5.1 Partial types

Partial types allow a single class, struct, or interface to be declared in multiple parts and usually across multiple source files. This is typically used with code generators. Multiple code generators can contribute different parts to the same type, and these can be further augmented by manually written code. The various parts are combined by the compiler and act as if they were all declared together.

Partial types are declared by adding the `partial` modifier to the type declaration. This must be present in every part. The following listing shows an example with two parts and demonstrates how a method declared in one part can be used in a different part.

Listing 2.22 A simple partial class

```
partial class PartialDemo
{
    public static void MethodInPart1()
    {
        MethodInPart2();           ← [Uses method declared
    }                                in second part
}

partial class PartialDemo
{
    private static void MethodInPart2()   ← [Method used
    {
        Console.WriteLine("In MethodInPart2");
    }
}
```

If the type is generic, every part has to declare the same set of type parameters with the same names, although if multiple declarations constrain the same type parameter, those constraints must be the same. Different parts can contribute different interfaces that a type implements, and the implementation doesn't need to be in the part that specifies the interface.

PARTIAL METHODS (C# 3)

C# 3 introduced an extra feature to partial types called *partial methods*. These are methods declared without a body in one part and then optionally implemented in another part. Partial methods are implicitly private and must be `void` with no `out` parameters. (It's fine to use `ref` parameters.) At compile time, only partial methods that have implementations are retained; if a partial method hasn't been implemented, all calls to it are removed. This sounds odd, but it allows generated code to provide optional hooks for manually written code to add extra behavior. It turns out to be useful indeed. The following listing provides an example with two partial methods, one of which is implemented and one of which isn't.

Listing 2.23 Two partial methods—one implemented, one not

```
partial class PartialMethodsDemo
{
    public PartialMethodsDemo()
    {
        OnConstruction();           ← Call to unimplemented
    }

    public override string ToString()
    {
        string ret = "Original return value";
        CustomizeToString(ref ret); ← Call to implemented
        return ret;
    }

    partial void OnConstruction();
    partial void CustomizeToString(ref string text);   | Partial method declarations
}

partial class PartialMethodsDemo
{
    partial void CustomizeToString(ref string text)      ← Partial method
    {
        text += " - customized!";
    }
}
```

In listing 2.23, the first part would most likely be generated code, thereby allowing for additional behavior on construction and when obtaining a string representation of the object. The second part corresponds to manually written code that doesn't need to customize construction but does want to change the string representation returned by `ToString()`. Even though the `CustomizeToString` method can't return a value directly, it can effectively pass information back to its caller with a `ref` parameter.

Because `OnConstruction` is never implemented, it's completely removed by the compiler. If a partial method with parameters is called, the arguments are never even evaluated when there's no implementation.

If you ever find yourself writing a code generator, I strongly encourage you to make it generate partial classes. You may also find it useful to create partial classes in purely handwritten code; I've used this to split tests for large classes into multiple source files for easy organization, for example.

2.5.2 Static classes

Static classes are classes declared with the `static` modifier. If you've ever found yourself writing utility classes composed entirely of static methods, those are prime candidates to be static classes. Static classes can't declare instance methods, properties, events, or constructors, but they can contain regular nested types.

Although it's perfectly valid to declare a regular class with only static members, adding the `static` modifier signals your intent in terms of how you expect the class to be used. The compiler knows that static classes can never be instantiated, so it prevents them from being used as either variable types or type arguments. The following listing gives a brief example of what's allowed and what's not.

Listing 2.24 Demonstration of static classes

```
static class StaticClassDemo
{
    public static void StaticMethod() { }

    public void InstanceMethod() { }

    public class RegularNestedClass
    {
        public void InstanceMethod() { }
    }
    ...
}

StaticClassDemo.StaticMethod();
```

The code in Listing 2.24 is annotated with several callouts explaining the rules of static classes:

- A callout points to the first `StaticMethod()` declaration with the text "Fine: static classes can declare static methods." and an arrow pointing to the method definition.
- A callout points to the second `InstanceMethod()` declaration with the text "Invalid: static classes can't declare instance methods." and an arrow pointing to the method definition.
- A callout points to the `RegularNestedClass` declaration with the text "Fine: static classes can declare regular nested types." and an arrow pointing to the class definition.
- A callout points to the inner `InstanceMethod()` declaration with the text "Fine: a regular type nested in a static class can declare an instance method." and an arrow pointing to the method definition.
- A callout points to the final line of code with the text "Fine: calling a static method from a static class" and an arrow pointing to the line.
- A callout points to the declaration of `localVariable` with the text "Invalid: can't declare a variable of a static class" and an arrow pointing to the variable declaration.
- A callout points to the declaration of `list` with the text "Invalid: can't use a static class as a type argument" and an arrow pointing to the variable declaration.

Static classes have additional special behavior in that extension methods (introduced in C# 3) can be declared only in non-nested, nongeneric, static classes.

2.5.3 Separate getter/setter access for properties

It's hard to believe, but in C# 1, a property had only a single access modifier that was used for both the getter and the setter, assuming both were present. C# 2 introduced the ability to make one accessor more private than the other by adding a modifier to that more-private accessor. This is almost always used to make the setter more private

than the getter, and by far the most common combination is to have a public getter and a private setter, like this:

```
private string text;

public string Text
{
    get { return text; }
    private set { text = value; }
}
```

In this example, any code that has access to the property setter could just set the field value directly, but in more complex situations, you may want to add validation or change notification. Using a property allows behavior like this to be encapsulated nicely. Although this could be put in a method instead, using a property feels more idiomatic in C#.

2.5.4 Namespace aliases

Namespaces are used to allow multiple types with the same name to be declared but in different namespaces. This avoids long and convoluted type names just for the sake of uniqueness. C# 1 already supported namespaces and even *namespace aliases* so you could make it clear which type you meant if you had a single piece of code that needed to use types with the same name from different namespaces. The following listing shows how one method can refer to the Button classes from both Windows Forms and ASP.NET Web Forms.

Listing 2.25 Namespace aliases in C# 1

```
using System;
using WinForms = System.Windows.Forms;
using WebForms = System.Web.UI.WebControls;

class Test
{
    static void Main()
    {
        Console.WriteLine(typeof(WinForms.Button));
        Console.WriteLine(typeof(WebForms.Button));
    }
}
```

Introduces namespace aliases
Uses the aliases to qualify a name

C# 2 extends the support for namespace aliases in three important ways.

NAMESPACE ALIAS QUALIFIER SYNTAX

The `WinForms.Button` syntax in listing 2.25 works fine so long as there isn't a type called `WinForms` as well. At that point, the compiler would treat `WinForms.Button` as an attempt to use a member called `Button` within the type `WinForms` instead of using the namespace alias. C# 2 solves this by introducing a new piece of syntax called a *namespace alias qualifier*, which is just a pair of colons. This is used only for namespace

aliases, thereby removing any ambiguity. Using namespace alias qualifiers, the `Main` method in listing 2.25 would become the following:

```
static void Main()
{
    Console.WriteLine(typeof(WinForms::Button));
    Console.WriteLine(typeof(WebForms::Button));
}
```

Resolving ambiguity is useful for more than just helping the compiler. More important, it helps anyone reading your code understand that the identifier before the `:` is expected to be a namespace alias, not a type name. I suggest using `::` anywhere you use a namespace alias.

THE GLOBAL NAMESPACE ALIAS

Although it's unusual to declare types in the global namespace in production code, it can happen. Prior to C# 2, there was no way of fully qualifying a reference to a type in the namespace. C# 2 introduces `global` as a namespace alias that always refers to the global namespace. In addition to referring to types in the global namespace, the global namespace alias can be used as a sort of "root" for fully qualified names, and this is how I've used it most often.

As an example, recently I was dealing with some code with a lot of methods using `DateTime` parameters. When another type called `DateTime` was introduced into the same namespace, that caused problems for these method declarations. Although I could've introduced a namespace alias for the `System` namespace, it was simpler to replace each method parameter type with `global::System.DateTime`. I find that namespace aliases in general, and particularly the global namespace alias, are especially useful when writing code generators or working with generated code where collisions are more likely to occur.

EXTERN ALIASES

So far I've been talking about naming collisions between multiple types with the same name but in different namespaces. What about a more worrying collision: two types with the same name in the same namespace but provided by different assemblies?

This is definitely a corner case, but it can come up, and C# 2 introduced *extern aliases* to handle it. Extern aliases are declared in source code without any specified association, like this:

```
extern alias FirstAlias;
extern alias SecondAlias;
```

In the same source code, you can then use the alias in `using` directives or writing fully qualified type names. For example, if you were using Json.NET but had an additional assembly that declared `Newtonsoft.Json.Linq JObject`, you could write code like this:

```
extern alias JsonNet;
extern alias JsonNetAlternative;
```

```

using JsonNet::Newtonsoft.Json.Linq;
using Alt JObject = JsonNetAlternative::Newtonsoft.Json.Linq.JObject;
...
JObject obj = new JObject();
Alt JObject alt = new Alt JObject(); ←

```

↳ **Uses the regular
Json.NET JObject type**

↳ **Uses the JObject type in
the alternative assembly**

That leaves one problem: associating each extern alias with an assembly. The mechanism for doing this is implementation specific. For example, it could be specified in project options or on the compiler command line.

I can't remember ever having to use extern aliases myself, and I'd normally expect them to be used as a stopgap solution while alternative approaches were being found to avoid the naming collision to start with. But I'm glad they exist to allow those temporary solutions.

2.5.5 Pragma directives

Pragma directives are implementation-specific directives that give extra information to the compiler. A pragma directive can't change the behavior of the program to contravene anything within the C# language specification, but it can do anything outside the scope of the specification. If the compiler doesn't understand a particular pragma directive, it can issue a warning but not an error. The syntax for pragma directives is simple: it's just `#pragma` as the first nonwhitespace part of a line followed by the text of the pragma directive.

The Microsoft C# compiler supports pragma directives for warnings and checksums. I've always seen checksum pragmas only in generated code, but warning pragmas are useful for disabling and reenabling specific warnings. For example, to disable warning CS0219 ("variable is assigned but its value is never used") for a specific piece of code, you might write this:

```

#pragma warning disable CS0219
int variable = CallSomeMethod();
#pragma warning restore CS0219

```

Until C# 6, warnings could be specified only using numbers. Roslyn makes the compiler pipeline more extensible, thereby allowing other packages to contribute warnings as part of the build. To accommodate this, the language was changed to allow a prefix (for example, CS for the C# compiler) to be specified as part of the warning identifier as well. I recommend always including the prefix (CS0219 rather than just 0219 in the preceding example) for clarity.

If you omit a specific warning identifier, *all* warnings will be disabled or restored. I've never used this facility, and I recommend against it in general. Usually, you want to fix warnings instead of disabling them, and disabling them on a blanket basis hides information about problems that might be lurking in your code.

2.5.6 Fixed-size buffers

Fixed-size buffers are another feature I've never used in production code. That doesn't mean you won't find them useful, particularly if you use interop with native code a lot.

Fixed-size buffers can be used only in unsafe code and only within structs. They effectively allocate a chunk of memory inline within the struct using the `fixed` modifier. The following listing shows a trivial example of a struct that represents 16 bytes of arbitrary data and two 32-bit integers to represent the major and minor versions of that data.

Listing 2.26 Using fixed-size buffers for a versioned chunk of binary data

```
unsafe struct VersionedData
{
    public int Major;
    public int Minor;
    public fixed byte Data[16];
}

unsafe static void Main()
{
    VersionedData versioned = new VersionedData();
    versioned.Major = 2;
    versioned.Minor = 1;
    versioned.Data[10] = 20;
}
```

I'd expect the size of a value of this struct type to be 24 bytes or possibly 32 bytes if the runtime aligned the fields to 8-byte boundaries. The important point is that all of the data is directly within the value; there's no reference to a separate byte array. This struct could be used for interoperability with native code or just used within regular managed code.

WARNING Although I provide a general warning about using sample code in this book, I feel compelled to give a more specific one for this example. To keep the code short, I haven't attempted to provide any encapsulation in this struct. It should be used only to get an impression of the syntax for fixed-size buffers.

IMPROVED ACCESS TO FIXED-SIZED BUFFERS IN FIELDS IN C# 7.3

Listing 2.26 demonstrated accessing a fixed-sized buffer via a local variable. If the `versioned` variable had been a field instead, accessing elements of `versioned`.`Data` would've required a `fixed` statement to create a pointer prior to C# 7.3. As of C# 7.3, you can access fixed-sized buffers in fields directly, although the code still needs to be in an unsafe context.

2.5.7 InternalsVisibleTo

The final feature for C# 2 is as much a framework and runtime feature as anything else. It isn't even mentioned in the language specification, although I'd expect any modern C# compiler to be aware of it. The framework exposes an attribute called

[`InternalsVisibleToAttribute`], which is an assembly-level attribute with a single parameter specifying another assembly. This allows internal members of the assembly containing the attribute to be used by the assembly specified in the attribute, as shown in the following example:

```
[assembly: InternalsVisibleTo("MyProduct.Test")]
```

When the assembly is signed, you need to include the public key in the assembly name. For example, in Noda Time I have this:

```
[assembly: InternalsVisibleTo("NodaTime.Test", PublicKey=0024...4669")]
```

The real public key is much longer than that, of course. Using this attribute with signed assemblies is never pretty, but you don't need to look at the code often. I've used the attribute in three kinds of situations, one of which I later regretted:

- Allowing a test assembly access to internal members to make testing easier
- Allowing tools (which are never published) access to internal members to avoid code duplication
- Allowing one library access to internal members in another closely related library

The last of these was a mistake. We're used to expecting that we can change internal code without worrying about versioning, but when internal code is exposed to another library that's versioned independently, it takes on the same versioning characteristics as public code. I don't intend to do that again.

For testing and tools, however, I'm a big fan of making the internals visible. I know there's testing dogma around testing only the public API surface, but often if you're trying to keep the public surface small, allowing your tests access to the internal code allows you to write much simpler tests, which means you're likely to write more of them.

Summary

- The changes in C# 2 made an enormous difference to the look and feel of idiomatic C#. Working without generics or nullable types is frankly horrible.
- Generics allow both types and methods to say more about the types in their API signatures. This promotes compile-time type safety without a lot of code duplication.
- Reference types have always had the ability to use a null value to express an absence of information. Nullable value types apply that idea to value types with support in the language, runtime, and framework to make them easy to work with.
- Delegates became easier to work with in C# 2, and method group conversions for regular methods and anonymous methods provide even more power and brevity.
- Iterators allow code to produce sequences that are lazily evaluated, which effectively pauses a method until the next value is requested.
- Not all features are huge. Small features such as partial types and static classes can still have a significant impact. Some of these won't affect every developer but will be vital for niche use cases.

C# 3: *LINQ and everything that comes with it*

This chapter covers

- Implementing trivial properties simply
- Initializing objects and collections more concisely
- Creating anonymous types for local data
- Using lambda expressions to build delegates and expression trees
- Expressing complex queries simply with query expressions

The new features of C# 2 were mostly independent of each other. Nullable value types depended on generics, but they were still separate features that didn't build toward a common goal.

C# 3 was different. It consisted of many new features, each of which was useful in its own right, but almost all of which built toward the larger goal of LINQ. This chapter shows each feature individually and then demonstrates how they fit together. The first feature we'll look at is the only one that has no direct relationship with LINQ.

3.1 Automatically implemented properties

Prior to C# 3, every property had to be implemented manually with bodies for the get and/or set accessors. The compiler was happy to provide an implementation for field-like events but not properties. That meant there were a lot of properties like this:

```
private string name;
public string Name
{
    get { return name; }
    set { name = value; }
}
```

Formatting would vary by code style, but whether the property was one long line, 11 short ones, or five lines in between (as in the preceding example), it was always just noise. It was a very long-winded way of expressing the intention to have a field and expose its value to callers via a property.

C# 3 made this much simpler by using *automatically implemented properties* (often referred to as *automatic properties* or even *autoprops*). These are properties with no accessor bodies; the compiler provides the implementation. The whole of the preceding code can be replaced with a single line:

```
public string Name { get; set; }
```

Note that there's no field declaration in the source code now. There's still a field, but it's created for you automatically by the compiler and given a name that can't be referred to anywhere in the C# code.

In C# 3, you can't declare read-only automatically implemented properties, and you can't provide an initial value at the point of declaration. Both of those features were introduced (finally!) in C# 6 and are described in section 8.2. Before C# 6, it was a reasonably common practice to fake read-only properties by giving them a private set accessor like this:

```
public string Name { get; private set; }
```

The introduction of automatically implemented properties in C# 3 had a huge effect in reducing boilerplate code. They're useful only when the property simply fetches and sets the field value, but that accounts for a large proportion of properties in my experience.

As I mentioned, automatically implemented properties don't directly contribute to LINQ. Let's move on to the first feature that does: implicit typing for arrays and local variables.

3.2 Implicit typing

In order to be as clear as possible about the features introduced in C# 3, I need to define a few terms first.

3.2.1 Typing terminology

Many terms are used to describe the way programming languages interact with their type system. Some people use the terms *weakly typed* and *strongly typed*, but I try to avoid those because they're not clearly defined and mean different things to different developers. Two other aspects have more consensus: static/dynamic typing and explicit/implicit typing. Let's look at each of those in turn.

STATIC AND DYNAMIC TYPING

Languages that are *statically typed* are typically compiled languages; the compiler is able to determine the type of each expression and check that it's used correctly. For example, if you make a method call on an object, the compiler can use the type information to check that there's a suitable method to call based on the type of the expression the method is called on, the name of the method, and the number and types of the arguments. Determining the meaning of something like a method call or field access is called *binding*. Languages that are *dynamically typed* leave all or most of the binding to execution time.

NOTE As you'll see in various places, some expressions in C# don't have a type when considered in source code, such as the null literal. But the compiler always works out a type based on the context in which the expression is used, at which point that type can be used for checking how the expression is used.

Aside from the dynamic binding introduced in C# 4 (and described in chapter 4), C# is a statically typed language. Even though the choice of which implementation of a virtual method should be executed depends on the execution-time type of the object it's called on, the binding process of determining the method signature all happens at compile time.

EXPLICIT AND IMPLICIT TYPING

In a language that's *explicitly typed*, the source code specifies all the types involved. This could be for local variables, fields, method parameters, or method return types, for example. A language that's *implicitly typed* allows the developer to omit the types from the source code so some other mechanism (whether it's a compiler or something at execution time) can infer which type is meant based on other context.

C# is mostly explicitly typed. Even before C# 3, there was some implicit typing, such as type inference for generic type arguments as you saw in section 2.1.4. Arguably, the presence of implicit conversions (such as `int` to `long`) make the language less explicitly typed, too.

With those different aspects of typing separated, you can look at the C# 3 features around implicit typing. We'll start with implicitly typed local variables.

3.2.2 Implicitly typed local variables (`var`)

Implicitly typed local variables are variables declared with the contextual keyword `var` instead of the name of a type, such as the following:

```
var language = "C#";
```

The result of declaring a local variable with `var` instead of with the name of a type is still a local variable with a known type; the only difference is that the type is inferred by the compiler from the compile-time type of the value assigned to it. The preceding code will generate the exact same result as this:

```
string language = "C#";
```

TIP When C# 3 first came out, a lot of developers avoided `var` because they thought it would remove a lot of compile-time checks or lead to execution-time performance problems. It doesn't do that at all; it only infers the type of the local variable. After the declaration, the variable acts exactly as if it had been declared with an explicit type name.

The way the type is inferred leads to two important rules for implicitly typed local variables:

- The variable must be initialized at the point of declaration.
- The expression used to initialize the variable must have a type.

Here's some invalid code to demonstrate these rules:

<pre><code>var x; x = 10;</code></pre>	No initial value provided
<pre><code>var y = null;</code></pre>	Initial value has no type.

It would've been possible to avoid these rules in some cases by analyzing all the assignments performed to the variable and inferring the type from those. Some languages do that, but the C# language designers preferred to keep the rules as simple as possible.

Another restriction is that `var` can be used for only local variables. Many times I've longed for implicitly typed fields, but they're still not available (as of C# 7.3, anyway).

In the preceding example, there was little benefit, if any, in using `var`. The explicit declaration is feasible and just as readable. There are generally three reasons for using `var`:

- When the type of the variable can't be named because it's anonymous. You'll look at anonymous types in section 3.4. This is the LINQ-related part of the feature.
- When the type of the variable has a long name and can easily be inferred by a human reader based on the expression used to initialize it.
- When the *precise* type of the variable isn't particularly important, and the expression used to initialize it gives enough information to anyone reading the code.

I'll save examples of the first bullet point for section 3.4, but it's easy to show the second. Suppose you want to create a dictionary that maps a name to a list of decimal values. You can do that with an explicitly typed variable:

```
Dictionary<string, List<decimal>> mapping =
    new Dictionary<string, List<decimal>>();
```

That's really ugly. I had to wrap it on two lines just to make it fit on the page, and there's a lot of duplication. That duplication can be entirely avoided by using `var`:

```
var mapping = new Dictionary<string, List<decimal>>();
```

This expresses the same amount of information in less text, so there's less to distract you from other code. Of course, this works only when you want the type of the variable to be exactly the type of the initialization expression. If you wanted the type of the mapping variable to be `IDictionary<string, List<decimal>>`—the interface instead of the class—then `var` wouldn't help. But for local variables, that sort of separation between interface and implementation is usually less important.

When I wrote the first edition of *C# in Depth*, I was wary of implicitly typed local variables. I rarely used them outside LINQ, apart from when I was calling a constructor directly, as in the preceding example. I was worried that I wouldn't be able to easily work out the type of the variable when just reading the code.

Ten years later, that caution has mostly gone. I use `var` for almost all my local variables in test code and extensively in production code, too. My fears weren't realized; in almost every case, I'm easily able to infer what the type should be just by inspection. Where that isn't the case, I'll happily use an explicit declaration instead.

I don't claim to be entirely consistent about this, and I'm certainly not dogmatic. Because explicitly typed variables generate the exact same code as implicitly typed variables, it's fine to change your mind later in either direction. I suggest you discuss this with the other people who'll work with your code the most (whether those are colleagues or open source collaborators), get a sense of everyone's comfort level, and try to abide by that. The other aspect of implicit typing in C# 3 is somewhat different. It's not directly related to `var`, but it has the same aspect of removing a type name to let the compiler infer it.

3.2.3 Implicitly typed arrays

Sometimes you need to create an array without populating it and keep all the elements with their default values. The syntax for that hasn't changed since C# 1; it's always something like this:

```
int[] array = new int[10];
```

But you often want to create an array with specific initial content. Before C# 3, there were two ways of doing this:

```
int[] array1 = { 1, 2, 3, 4, 5};
int[] array2 = new int[] { 1, 2, 3, 4, 5};
```

The first form of this is valid only when it's part of a variable declaration that specifies the array type. This is invalid, for example:

```
int[] array;
array = { 1, 2, 3, 4, 5 };           ←— Invalid
```

The second form is always valid, so the second line in the preceding example could've been as follows:

```
array = new int[] { 1, 2, 3, 4, 5 };
```

C# 3 introduced a third form in which the type of the array is implicit based on the content:

```
array = new[] { 1, 2, 3, 4, 5 };
```

This can be used anywhere, so long as the compiler is able to infer the array element type from the array elements specified. It also works with multidimensional arrays, as in the following example:

```
var array = new[,] { { 1, 2, 3 }, { 4, 5, 6 } };
```

The next obvious question is how the compiler infers that type. As is so often the case, the *precise* details are complex in order to handle all kinds of corner cases, but the simplified sequence of steps is as follows:

- 1 Find a set of *candidate types* by considering the type of each array element that has a type.
- 2 For each candidate type, check whether every array element has an implicit conversion to that type. Remove any candidate type that doesn't meet this condition.
- 3 If there's exactly one type left, that's the inferred element type, and the compiler creates an appropriate array. Otherwise (if there are no types or more than one type left), a compile-time error occurs.

The array element type must be the type of one of the expressions in the array initializer. There's no attempt to find a common base class or a commonly implemented interface. Table 3.1 gives some examples that illustrate the rules.

Table 3.1 Examples of type inference for implicitly typed arrays

Expression	Result	Notes
new[] { 10, 20 }	int[]	All elements are of type int.
new[] { null, null }	Error	No elements have types.
new[] { "xyz", null }	string[]	Only candidate type is string, and the null literal can be converted to string.
new[] { "abc", new object() }	object[]	Candidate types of string and object; implicit conversion from string to object but not vice versa.

Table 3.1 Examples of type inference for implicitly typed arrays (continued)

Expression	Result	Notes
<code>new[] { 10, new DateTime() }</code>	Error	Candidate types of <code>int</code> and <code>DateTime</code> but no conversion from either to the other.
<code>new[] { 10, null }</code>	Error	Only candidate type is <code>int</code> , but there's no conversion from <code>null</code> to <code>int</code> .

Implicitly typed arrays are mostly a convenience to reduce the source code required except for anonymous types, where the array type can't be stated explicitly even if you want to. Even so, they're a convenience I'd definitely miss now if I had to work without them.

The next feature continues the theme of making it simpler to create and initialize objects, but in a different way.

3.3 Object and collection initializers

Object initializers and *collection initializers* make it easy to create new objects or collections with initial values, just as you can create and populate an array in a single expression. This functionality is important for LINQ because of the way queries are translated, but it turns out to be extremely useful elsewhere, too. It does require types to be mutable, which can be annoying if you're trying to write code in a functional style, but where you *can* apply it, it's great. Let's look at a simple example before diving into the details.

3.3.1 Introduction to object and collection initializers

As a *massively* oversimplified example, let's consider what an order in an e-commerce system might look like. The following listing shows three classes to model an order, a customer, and a single item within an order.

Listing 3.1 Modeling an order in an e-commerce system

```
public class Order
{
    private readonly List<OrderItem> items = new List<OrderItem>();

    public string OrderId { get; set; }
    public Customer Customer { get; set; }
    public List<OrderItem> Items { get { return items; } }
}

public class Customer
{
    public string Name { get; set; }
    public string Address { get; set; }
}

public class OrderItem
```

```
{
    public string ItemId { get; set; }
    public int Quantity { get; set; }
}
```

How do you create an order? Well, you need to create an instance of `Order` and assign to its `OrderId` and `Customer` properties. You can't assign to the `Items` property, because it's read-only. Instead, you can add items to the list it returns. The following listing shows how you might do this if you didn't have object and collection initializers and couldn't change the classes to make things simpler.

Listing 3.2 Creating and populating an order without object and collection initializers

```
var customer = new Customer();
customer.Name = "Jon";
customer.Address = "UK";                                | Creates the Customer

var item1 = new OrderItem();
item1.ItemId = "abcd123";
item1.Quantity = 1;                                     | Creates the first OrderItem

var item2 = new OrderItem();
item2.ItemId = "fghi456";
item2.Quantity = 2;                                     | Creates the second OrderItem

var order = new Order();
order.OrderId = "xyz";
order.Customer = customer;
order.Items.Add(item1);
order.Items.Add(item2);                                | Creates the order
```

This code could be simplified by adding constructors to the various classes to initialize properties based on the parameters. Even with object and collection initializers available, that's what I'd do. But for the sake of brevity, I'm going to ask you to trust me that it's not always feasible, for all kinds of reasons. Aside from anything else, you don't always control the code for the classes you're using. Object and collection initializers make it much simpler to create and populate our order, as shown in the following listing.

Listing 3.3 Creating and populating an order with object and collection initializers

```
var order = new Order
{
    OrderId = "xyz",
    Customer = new Customer { Name = "Jon", Address = "UK" },
    Items =
    {
        new OrderItem { ItemId = "abcd123", Quantity = 1 },
        new OrderItem { ItemId = "fghi456", Quantity = 2 }
    }
};
```

I can't speak for everyone, but I find listing 3.3 much more readable than listing 3.2. The structure of the object becomes apparent in the indentation, and less repetition occurs. Let's look more closely at each part of the code.

3.3.2 Object initializers

Syntactically, an object initializer is a sequence of *member initializers* within braces. Each member initializer is of the form `property = initializer-value`, where `property` is the name of the field or property being initialized and `initializer-value` is an expression, a collection initializer, or another object initializer.

NOTE Object initializers are most commonly used with properties, and that's how I've described them in this chapter. Fields don't have accessors, but the obvious equivalents apply: reading the field instead of calling a get accessor and writing the field instead of calling a set accessor.

Object initializers can be used only as part of a constructor call or another object initializer. The constructor call can specify arguments as usual, but if you don't want to specify any arguments, you don't need an argument list at all, so you can omit the `()`. A constructor call without an argument list is equivalent to supplying an empty argument list. For example, these two lines are equivalent:

```
Order order = new Order() { OrderId = "xyz" };
Order order = new Order { OrderId = "xyz" };
```

You can omit the constructor argument list only if you provide an object or collection initializer. This is invalid:

```
Order order = new Order;           ← Invalid
```

An object initializer simply says how to initialize each of the properties it mentions in its member initializers. If the `initializer-value` part (the part to the right of the `=` sign) is a normal expression, that expression is evaluated, and the value is passed to the property set accessor. That's how most of the object initializers in listing 3.3 work. The `Items` property uses a *collection initializer*, which you'll see shortly.

If `initializer-value` is another object initializer, the set accessor is never called. Instead, the get accessor is called, and then the nested object initializer is applied to the value returned by the property. As an example, listing 3.4 creates an `HttpClient` and modifies the set of default headers that are sent with each request. The code sets the `From` and `Date` headers, which I chose only because they're the simplest ones to set.

Listing 3.4 Modifying default headers on a new HttpClient with a nested object initializer

```
HttpClient client = new HttpClient
{
    DefaultRequestHeaders =
    {
```



Property get accessor called
for DefaultRequestHeaders

```

    From = "user@example.com",
    Date = DateTimeOffset.UtcNow
}
};

Property set accessor  
called for From ←
Property set accessor  
called for Date ←

```

The code in listing 3.4 is equivalent to the following code:

```

HttpClient client = new HttpClient();
var headers = client.DefaultRequestHeaders;
headers.From = "user@example.com";
headers.Date = DateTimeOffset.UtcNow;

```

A single object initializer can include a mixture of nested object initializers, collection initializers, and normal expressions in the sequence of member initializers. Speaking of collection initializers, let's look at those now.

3.3.3 Collection initializers

Syntactically, a collection initializer is a comma-separated list of *element initializers* in curly braces. Each element initializer is either a single expression or a comma-separated list of expressions also in curly braces. Collection initializers can be used only as part of a constructor call or part of an object initializer. Further restrictions exist on the types they can be used with, which we'll come to shortly. In listing 3.3, you saw a collection initializer being used as part of an object initializer. Here's the listing again with the collection initializer highlighted in bold:

```

var order = new Order
{
    OrderId = "xyz",
    Customer = new Customer { Name = "Jon", Address = "UK" },
    Items =
    {
        new OrderItem { ItemId = "abcd123", Quantity = 1 },
        new OrderItem { ItemId = "fghi456", Quantity = 2 }
    }
};

```

Collection initializers might be more commonly used when creating new collections, though. For example, this line declares a new variable for a list of strings and populates the list:

```
var beatles = new List<string> { "John", "Paul", "Ringo", "George" };
```

The compiler compiles that into a constructor call followed by a sequence of calls to an Add method:

```

var beatles = new List<string>();
beatles.Add("John");
beatles.Add("Paul");
beatles.Add("Ringo");
beatles.Add("George");

```

But what if the collection type you're using doesn't have an Add method with a single parameter? That's where element initializers with braces come in. After `List<T>`, the second most common generic collection is probably `Dictionary< TKey, TValue >` with an `Add(key, value)` method. A dictionary can be populated with a collection initializer like this:

```
var releaseYears = new Dictionary<string, int>
{
    { "Please please me", 1963 },
    { "Revolver", 1966 },
    { "Sgt. Pepper's Lonely Hearts Club Band", 1967 },
    { "Abbey Road", 1970 }
};
```

The compiler treats each element initializer as a separate Add call. If the element initializer is a simple one without braces, the value is passed as a single argument to Add. That's what happened for the elements in our `List<string>` collection initializer.

If the element initializer uses braces, it's still treated as a single call to Add, but with one argument for each expression within the braces. The preceding dictionary example is effectively equivalent to this:

```
var releaseYears = new Dictionary<string, int>();
releaseYears.Add("Please please me", 1963);
releaseYears.Add("Revolver", 1966);
releaseYears.Add("Sgt. Pepper's Lonely Hearts Club Band", 1967);
releaseYears.Add("Abbey Road", 1970);
```

Overload resolution then proceeds as normal to find the most appropriate Add method, including performing type inference if there are any generic Add methods.

Collection initializers are valid only for types that implement `IEnumerable`, although they don't have to implement `IEnumerable<T>`. The language designers looked at the types in the framework that had Add methods and determined that the best way of separating them into collections and noncollections was to look at whether they implemented `IEnumerable`. As an example of why that's important, consider the `DateTime.Add(TimeSpan)` method. The `DateTime` type clearly isn't a collection, so it'd be odd to be able to write this:

```
DateTime invalid = new DateTime(2020, 1, 1) { TimeSpan.FromDays(10) };    ↗
```

Invalid

The compiler never uses the implementation of `IEnumerable` when compiling a collection initializer. I've sometimes found it convenient to create types in test projects with Add methods and an implementation of `IEnumerable` that just throws a `NotImplementedException`. This can be useful for constructing test data, but I don't advise doing it in production code. I'd appreciate an attribute that let me express the idea that this type should be usable for collection initializers without implementing `IEnumerable`, but I doubt that'll ever happen.

3.3.4 The benefits of single expressions for initialization

You may be wondering what all of this has to do with LINQ. I said that almost all the features in C# 3 built up to LINQ, so how do object and collection initializers fit into the picture? The answer is that other LINQ features require code to be expressible as a single expression. (For example, in a query expression, you can't write a `select` clause that requires multiple statements to produce the output for a given input.)

The ability to initialize new objects in a single expression isn't useful only for LINQ, however. It can also be important to simplify field initializers, method arguments, or even the operands in a conditional `? :` operator. I find it particularly useful for static field initializers to build up useful lookup tables, for example. Of course, the larger the initialization expression becomes, the more you may want to consider separating it out.

It's even recursively important to the feature itself. For example, if we couldn't use an object initializer to create our `OrderItem` objects, the collection initializer wouldn't be nearly as convenient to populate the `Order.Items` property.

In the rest of this book, whenever I refer to a new or improved feature as having a special case for a single expression (such as lambda expressions in section 3.5 or expression-bodied members in section 8.3), it's worth remembering that object and collection initializers immediately make that feature more useful than it'd be otherwise.

Object and collection initializers allow for more concise code to create an instance of a type and populate it, but they do require that you already have an appropriate type to construct. Our next feature, anonymous types, allows you to create objects without even declaring the type of the object beforehand. It's not *quite* as strange as it sounds.

3.4 Anonymous types

Anonymous types allow you to build objects that you can refer to in a statically typed way without having to declare a type beforehand. This sounds like types might be created dynamically at execution time, but the reality is a little more subtle than that. We'll look at what anonymous types look like in source code, how the compiler handles them, and a few of their limitations.

3.4.1 Syntax and basic behavior

The simplest way to explain anonymous types is to start with an example. The following listing shows a simple piece of code to create an object with `Name` and `Score` properties.

Listing 3.5 Anonymous type with Name and Score properties

```
var player = new
{
    Name = "Rajesh",
    Score = 3500
};
```

Creates an object of an
anonymous type with Name
and Score properties

```
Console.WriteLine("Player name: {0}", player.Name);
Console.WriteLine("Player score: {0}", player.Score);
```

Displays the
property values

This brief example demonstrates important points about anonymous types:

- The syntax is a little like object initializers but without specifying a type name; it's just new, open brace, properties, close brace. This is called an *anonymous object creation expression*. The property values can be nested anonymous object creation expressions.
- You're using var for the declaration of the player variable, because the type has no name for you to use instead of var. (The declaration would work if you used object instead, but it wouldn't be nearly as useful.)
- This code is still statically typed. Visual Studio can autocomplete the Name and Score properties of the player variable. If you ignore that and try to access a property that doesn't exist (if you try to use player.Points, for example), the compiler will raise an error. The property types are inferred from the values assigned to them; player.Name is a string property, and player.Score is an int property.

That's what anonymous types look like, but what are they used for? This is where LINQ comes in. When performing a query, whether that's using an SQL database as the underlying data store or using a collection of objects, it's common to want a specific shape of data that isn't the original type and may not have much meaning outside the query.

For example, suppose you're building a query using a set of people, each of which has expressed a favorite color. You might want the result to be a histogram: each entry in the resulting collection is the color and the number of people who chose that as their favorite. That type representing a favorite color and type isn't likely to be useful anywhere else, but it is useful in this specific context. Anonymous types allow us to express those one-off cases concisely without losing the benefits of static typing.

Comparison with Java anonymous classes

If you're familiar with Java, you may be wondering about the relationship between C#'s anonymous types and Java's anonymous classes. They sound like they'd be similar, but they differ greatly both in syntax and purpose.

Historically, the principal use for anonymous classes in Java was to implement interfaces or extend abstract classes to override just one or two methods. C#'s anonymous types don't allow you to implement an interface or derive from any class other than System.Object; their purpose is much more about data than executable code.

C# provides one extra piece of shorthand in anonymous object creation expressions where you're effectively copying a property or field from somewhere else and you're happy to use the same name. This syntax is called a *projection initializer*. To give an example, let's go back to our simplified e-commerce data model. You have three classes:

- Order—OrderId, Customer, Items
- Customer—Name, Address
- OrderItem—ItemId, Quantity

At some point in your code, you may want an object with all this information for a specific order item. If you have variables of the relevant types called `order`, `customer`, and `item`, you can easily use an anonymous type to represent the flattened information:

```
var flattenedItem = new
{
    order.OrderId,
    CustomerName = customer.Name,
    customer.Address,
    item.ItemId,
    item.Quantity
};
```

In this example, every property except `CustomerName` uses a projection initializer. The result is identical to this code, which specifies the property names in the anonymous type explicitly:

```
var flattenedItem = new
{
    OrderId = order.OrderId,
    CustomerName = customer.Name,
    Address = customer.Address,
    ItemId = item.ItemId,
    Quantity = item.Quantity
};
```

Projection initializers are most useful when you're either performing a query and want to select only a subset of properties or to combine properties from multiple objects into one. If the name you want to give the property in the anonymous type is the same as the name of the field or property you're copying from, the compiler can infer that name for you. So instead of writing this

```
SomeProperty = variable.SomeProperty
```

you can just write this:

```
variable.SomeProperty
```

Projection initializers can significantly reduce the amount of duplication in your source code if you're copying multiple properties. It can easily make the difference between an expression being short enough to keep on one line or long enough to merit a separate line per property.

Refactoring and projection initializers

Although it's accurate to say that the results of the two preceding listings are the same, that doesn't mean they behave identically in other ways. Consider a rename of the `Address` property to `CustomerAddress`.

In the version with projection initializers, the property name in the anonymous type would change too. In the version with the explicit property name, it wouldn't. That's rarely an issue in my experience, but it's worth being aware of as a difference.

I've described the syntax of anonymous types, and you know the resulting objects have properties you can use as if they were normal types. But what's going on behind the scenes?

3.4.2 *The compiler-generated type*

Although the type never appears in source code, the compiler does generate a type. There's no magic for the runtime to contend with; it just sees a type that happens to have a name that would be invalid in C#. That type has a few interesting aspects to it. Some are guaranteed by the specification; others aren't. When using the Microsoft C# compiler, an anonymous type has the following characteristics:

- It's a class (guaranteed).
- Its base class is `object` (guaranteed).
- It's sealed (not guaranteed, although it would be hard to see how it would be useful to make it unsealed).
- The properties are all read-only (guaranteed).
- The constructor parameters have the same names as the properties (not guaranteed; can be useful for reflection occasionally).
- It's internal to the assembly (not guaranteed; can be irritating when working with dynamic typing).
- It overrides `GetHashCode()` and `Equals()` so that two instances are equal only if all their properties are equal. (It handles properties being null.) The fact that these methods are overridden is guaranteed, but the precise way of computing the hash code isn't.
- It overrides `ToString()` in a helpful way and lists the property names and their values. This isn't guaranteed, but it is super helpful when diagnosing issues.
- The type is generic with one type parameter for each property. Multiple anonymous types with the same property names but different property types will use different type arguments for the same generic type. This isn't guaranteed and could easily vary by compiler.
- If two anonymous object creation expressions use the same property names in the same order with the same property types in the same assembly, the result is guaranteed to be two objects of the same type.

The last point is important for variable reassignment and for implicitly typed arrays using anonymous types. In my experience, it's relatively rare that you want to reassign

a variable initialized with an anonymous type, but it's nice that it's feasible. For example, this is entirely valid:

```
var player = new { Name = "Pam", Score = 4000 };
player = new { Name = "James", Score = 5000 };
```

Likewise, it's fine to create an array by using anonymous types using the implicitly typed array syntax described in section 3.2.3:

```
var players = new[]
{
    new { Name = "Priti", Score = 6000 },
    new { Name = "Chris", Score = 7000 },
    new { Name = "Amanda", Score = 8000 },
};
```

Note that the properties must have the same names and types and be in the same order for two anonymous object creation expressions to use the same type. For example, this would be invalid because the order of properties in the second array element is different from the others:

```
var players = new[]
{
    new { Name = "Priti", Score = 6000 },
    new { Score = 7000, Name = "Chris" },
    new { Name = "Amanda", Score = 8000 },
};
```

Although each array element is valid individually, the type of the second element stops the compiler from inferring the array type. The same would be true if you added an extra property or changed the type of one of the properties.

Although anonymous types are useful within LINQ, that doesn't make this feature the right tool for every problem. Let's look briefly at places you may not want to use them.

3.4.3 Limitations

Anonymous types are great when you want a localized representation of just data. By *localized*, I mean that the data shape you're interested in is relevant only within that specific method. As soon as you want to represent the same shape in multiple places, you need to look for a different solution. Although it's possible to return instances of anonymous types from methods or accept them as parameters, you can do so only by using either generics or the `object` type. The fact that the types are anonymous prevents you from expressing them in method signatures.

Until C# 7, if you wanted to use a common data structure in more than one method, you'd normally declare your own class or struct for it. C# 7 has introduced tuples, as you'll see in chapter 11, which can work as an alternative solution, depending on how much encapsulation you desire.

Speaking of encapsulation, anonymous types basically don't provide any. You can't place any validation in the type or add extra behavior to it. If you find yourself wanting to do so, that's a good indication that you should probably be creating your own type instead.

Finally, I mentioned earlier that using anonymous types across assemblies via C# 4's dynamic typing is made more difficult because the types are internal. I've usually seen this attempted in MVC web applications where the model for a page may be built using anonymous types and then accessed in the view using the `dynamic` type (which you'll look at in chapter 4). This works if either the two pieces of code are in the same assembly or the assembly containing the model code has made its internal members visible to the assembly containing the view code using `[InternalsVisibleTo]`. Depending on the framework you're using, it may be awkward to arrange for either of these to be true. Given the benefits of static typing anyway, I generally recommend declaring the model as a regular type instead. It's more up-front work than using an anonymous type but is likely to save you time in the long term.

NOTE Visual Basic has anonymous types too, but they don't behave in quite the same way. In C#, all properties are used in determining equality and hash codes, and they're all read-only. In VB, only properties declared with the `Key` modifier behave like that. Nonkey properties are read/write and don't affect equality or hash codes.

We're about halfway through the C# 3 features, and so far they've all had to do with data. The next features focus more on executable code, first with lambda expressions and then extension methods.

3.5 Lambda expressions

In chapter 2, you saw how anonymous methods made it much easier to create delegate instances by including their code inline like this:

```
Action<string> action = delegate(string message)
{
    Console.WriteLine("In delegate: {0}", message);
};
action("Message");
```

Creates delegate using an anonymous method

Invokes the delegate

Lambda expressions were introduced in C# 3 to make this even more concise. The term *anonymous function* is used to refer to both anonymous methods and lambda expressions. I'll use it at various points in the rest of this book, and it's widely used in the C# specification.

NOTE The name *lambda expressions* comes from lambda calculus, a field of mathematics and computer science started by Alonzo Church in the 1930s. Church used the Greek lambda character (λ) in his notation for functions, and the name stuck.

There are various reasons that it was useful for the language designers to put so much effort into streamlining delegate instance creation, but LINQ is the most important one. When you look at query expressions in section 3.7, you'll see that they're effectively translated into code that uses lambda expressions. You can use LINQ without using query expressions, though, and that almost always involves using lambda expressions directly in your source code.

First, we'll look at the syntax for lambda expressions and then some of the details of how they behave. Finally, we'll talk about *expression trees* that represent code as data.

3.5.1 Lambda expression syntax

The basic syntax for lambda expressions is always of this form:

parameter-list => *body*

Both the parameter list and the body, however, have multiple representations. In its most explicit form, the parameter list for a lambda expression looks like a normal method or anonymous method parameter list. Likewise, the body of a lambda expression can be a block: a sequence of statements all within a pair of curly braces. In this form, the lambda expression looks similar to an anonymous method:

```
Action<string> action = (string message) =>
{
    Console.WriteLine("In delegate: {0}", message);
};
action("Message");
```

So far, this doesn't look much better; you've traded the `delegate` keyword for `=>`, but that's all. But special cases allow the lambda expression to become shorter.

Let's start by making the body more concise. A body that consists of just a return statement or a single expression can be reduced to that single expression. The `return` keyword is removed if there was one. In the preceding example, the body of our lambda expression was just a method invocation, so you can simplify it:

```
Action<string> action =
    (string message) => Console.WriteLine("In delegate: {0}", message);
```

You'll look at an example returning a value shortly. Lambda expressions shortened like this are said to have *expression bodies*, whereas lambda expressions using braces are said to have *statement bodies*.

Next, you can make the parameter list shorter if the compiler can infer the parameter types based on the type you're attempting to convert the lambda expression to. Lambda expressions don't have a type but are convertible to compatible delegate types, and the compiler can often infer the parameter type as part of that conversion.

For example, in the preceding code, the compiler knows that an `Action<string>` has a single parameter of type `string`, so it's capable of inferring

that parameter type. When the compiler can infer the parameter type, you can omit it. Therefore, our example can be shortened:

```
Action<string> action =
    (message) => Console.WriteLine("In delegate: {0}", message);
```

Finally, if the lambda expression has exactly one parameter, and that parameter's type is inferred, the parentheses can be dropped from the parameter list:

```
Action<string> action =
    message => Console.WriteLine("In delegate: {0}", message);
```

Now let's look at a couple of examples that return values. In each case, you'll apply every step you can to make it shorter. First, you'll construct a delegate to multiply two integers together and return the result:

```
Func<int, int, int> multiply =
    (int x, int y) => { return x * y; };           | Longest form
Func<int, int, int> multiply = (int x, int y) => x * y;   |<-----| Uses an
Func<int, int, int> multiply = (x, y) => x * y;       |<-----| Infers parameter types
(Two parameters, so you can't remove parentheses)
```

Next, you'll use a delegate to take the length of a string, multiply that length by itself, and return the result:

```
Func<string, int> squareLength = (string text) =>           |<-----| Longest
{                                                               | form
    int length = text.Length;
    return length * length;
};

Func<string, int> squareLength = (text) =>                   |<-----| Infers
{                                                               | parameter type
    int length = text.Length;
    return length * length;
};

Func<string, int> squareLength = text =>                     |<-----| Removes parentheses
{                                                               | for single parameter
    int length = text.Length;
    return length * length;
};
(Can't do anything else immediately; body has two statements)
```

If you were happy to evaluate the `Length` property twice, you could reduce this second example:

```
Func<string, int> squareLength = text => text.Length * text.Length;
```

That's not the same kind of change as the others, though; that's changing the *behavior* (however slightly) rather than just the syntax. It may seem odd to have all of these

special cases, but in practice all of them apply in a large number of cases, particularly within LINQ. Now that you understand the syntax, you can start looking at the behavior of the delegate instance, particularly in terms of any variables it has captured.

3.5.2 Capturing variables

In section 2.3.2, when I described captured variables in anonymous methods, I promised that we'd return to the topic in the context of lambda expressions. This is probably the most confusing part of lambda expressions. It's certainly been the cause of lots of Stack Overflow questions.

To create a delegate instance from a lambda expression, the compiler converts the code in the lambda expression to a method somewhere. The delegate can then be created at execution time exactly as if you had a method group. This section shows the kind of transformation the compiler performs. I've written this as if the compiler translates the source code into more source code that doesn't contain lambda expressions, but of course the compiler never needs that translated source code. It can just emit the appropriate IL.

Let's start with a recap of what counts as a captured variable. Within a lambda expression, you can use any variable that you'd be able to use in regular code at that point. That could be a static field, an instance field (if you're writing the lambda expression within an instance method¹), the `this` variable, method parameters, or local variables. All of these are captured variables, because they're variables declared outside the immediate context of the lambda expression. Compare that with parameters to the lambda expression or local variables declared within the lambda expression; those aren't captured variables. The following listing shows a lambda expression that captures various variables. You'll then look at how the compiler handles that code.

Listing 3.6 Capturing variables in a lambda expression

```
class CapturedVariablesDemo
{
    private string instanceField = "instance field";

    public Action<string> CreateAction(string methodParameter)
    {
        string methodLocal = "method local";
        string uncaptured = "uncaptured local";

        Action<string> action = lambdaParameter =>
        {
            string lambdaLocal = "lambda local";
            Console.WriteLine("Instance field: {0}", instanceField);
            Console.WriteLine("Method parameter: {0}", methodParameter);
            Console.WriteLine("Method local: {0}", methodLocal);
        };
    }
}
```

¹ You can write lambda expressions in constructors, property accessors, and so on as well, but for the sake of simplicity, I'll assume you're writing them in methods.

```

        Console.WriteLine("Lambda parameter: {0}", lambdaParameter);
        Console.WriteLine("Lambda local: {0}", lambdaLocal);
    };
    methodLocal = "modified method local";
    return action;
}
}

```

In other code

```

var demo = new CapturedVariablesDemo();
Action<string> action = demo.CreateAction("method argument");
action("lambda argument");

```

Lots of variables are involved here:

- `instanceField` is an instance field in the `CapturedVariablesDemo` class and is captured by the lambda expression.
- `methodParameter` is a parameter in the `CreateAction` method and is captured by the lambda expression.
- `methodLocal` is a local variable in the `CreateAction` method and is captured by the lambda expression.
- `uncaptured` is a local variable in the `CreateAction` method, but it's never used by the lambda expression, so it's not captured by it.
- `lambdaParameter` is a parameter in the lambda expression itself, so it isn't a captured variable.
- `lambdaLocal` is a local variable in the lambda expression, so it isn't a captured variable.

It's important to understand that the lambda expression captures the variables themselves, *not* the values of the variables at the point when the delegate is created.² If you modified any of the captured variables between the time at which the delegate is created and when it's invoked, the output would reflect those changes. Likewise, the lambda expression can change the value of the captured variables. How does the compiler make all of that work? How does it make sure all those variables are still available to the delegate when it's invoked?

IMPLEMENTING CAPTURED VARIABLES WITH A GENERATED CLASS

There are three broad cases to consider:

- If no variables are captured at all, the compiler can create a static method. No extra context is required.
- If the only variables captured are instance fields, the compiler can create an instance method. Capturing one instance field is equivalent to capturing 100 of them, because you need access only to `this`.

² I will repeat this multiple times, for which I make no apology. If you're new to captured variables, this can take a while to get used to.

- If local variables or parameters are captured, the compiler creates a private nested class to contain that context and then an instance method in that class containing the lambda expression code. The method containing the lambda expression is changed to use that nested class for every access to the captured variables.

Implementation details may vary

You may see some variation in what I've described. For example, with a lambda expression with no captured variables, the compiler may create a nested class with a single instance instead of a static method. There can be subtle differences in the efficiency of executing delegates based on exactly how they're created. In this section, I've described the minimum work that the compiler *must* do in order to make captured variables available. It can introduce more complexity if it wants to.

The last case is obviously the most complex one, so we'll focus on that. Let's start with listing 3.6. As a reminder, here's the method that creates the lambda expression; I've omitted the class declaration for brevity:

```
public Action<string> CreateAction(string methodParameter)
{
    string methodLocal = "method local";
    string uncaptured = "uncaptured local";

    Action<string> action = lambdaParameter =>
    {
        string lambdaLocal = "lambda local";
        Console.WriteLine("Instance field: {0}", instanceField);
        Console.WriteLine("Method parameter: {0}", methodParameter);
        Console.WriteLine("Method local: {0}", methodLocal);
        Console.WriteLine("Lambda parameter: {0}", lambdaParameter);
        Console.WriteLine("Lambda local: {0}", lambdaLocal);
    };
    methodLocal = "modified method local";
    return action;
}
```

As I described before, the compiler creates a private nested class for the extra context it'll need and then an instance method in that class for the code in the lambda expression. The context is stored in instance variables of the nested class. In our case, that means the following:

- A reference to the original instance of `CapturedVariablesDemo` so that you can access `instanceField` later
- A string variable for the captured method parameter
- A string variable for the captured local variable

The following listing shows the nested class and how it's used by the `CreateAction` method.

Listing 3.7 Translation of a lambda expression with captured variables

```

private class LambdaContext
{
    public CapturedVariablesDemoImpl originalThis;
    public string methodParameter;
    public string methodLocal;

    public void Method(string lambdaParameter)
    {
        string lambdaLocal = "lambda local";
        Console.WriteLine("Instance field: {0}",
            originalThis.instanceField);
        Console.WriteLine("Method parameter: {0}", methodParameter);
        Console.WriteLine("Method local: {0}", methodLocal);
        Console.WriteLine("Lambda parameter: {0}", lambdaParameter);
        Console.WriteLine("Lambda local: {0}", lambdaLocal);
    }
}

public Action<string> CreateAction(string methodParameter)
{
    LambdaContext context = new LambdaContext();
    context.originalThis = this;
    context.methodParameter = methodParameter;
    context.methodLocal = "method local";
    string uncaptured = "uncaptured local";

    Action<string> action = context.Method;
    context.methodLocal = "modified method local";
    return action;
}

```

Generated class to hold the captured variables

Captured variables

Body of lambda expression becomes an instance method.

Generated class is used for all captured variables.

Note how the `context.methodLocal` is modified near the end of the `CreateAction` method. When the delegate is finally invoked, it'll “see” that modification. Likewise, if the delegate modified any of the captured variables, each invocation would see the results of the previous invocations. This is just reinforcing that the compiler ensures that the variable is captured rather than a snapshot of its value.

In listings 3.6 and 3.7, you had to create only a single context for the captured variables. In the terminology of the specification, each of the local variables was instantiated only once. Let's make things a little more complicated.

MULTIPLE INSTANTIATIONS OF LOCAL VARIABLES

To make things a little simpler, you'll capture one local variable this time and no parameters or instance fields. The following listing shows a method to create a list of actions and then execute them one at a time. Each action captures a `text` variable.

Listing 3.8 Instantiating a local variable multiple times

```

static List<Action> CreateActions()
{
    List<Action> actions = new List<Action>();
    for (int i = 0; i < 5; i++)

```

```

    {
        string text = string.Format("message {0}", i);
        actions.Add(() => Console.WriteLine(text));
    }
    return actions;
}

```

Captures the variable in a lambda expression

Declares a local variable within the loop

In other code

```
List<Action> actions = CreateActions();
foreach (Action action in actions)
{
    action();
}
```

The fact that `text` is declared inside the loop is very important indeed. Each time you reach that declaration, the variable is instantiated. Each lambda expression captures a different instantiation of the variable. There are effectively five different `text` variables, each of which has been captured separately. They're completely independent variables. Although this code happens not to modify them after the initial assignment, it certainly could do so either inside the lambda expression or elsewhere within the loop. Modifying one variable would have no effect on the others.

The compiler models this behavior by creating a different instance of the generated type for each instantiation. Therefore, the `CreateAction` method of listing 3.8 could be translated into the following listing.

Listing 3.9 Creating multiple context instances, one for each instantiation

```

private class LambdaContext
{
    public string text;

    public void Method()
    {
        Console.WriteLine(text);
    }
}

static List<Action> CreateActions()
{
    List<Action> actions = new List<Action>();
    for (int i = 0; i < 5; i++)
    {
        LambdaContext context = new LambdaContext(); ← Creates a new context
        context.text = string.Format("message {0}", i); ← for each loop iteration
        actions.Add(context.Method()); ← Uses the context to
    }                                         create an action
    return actions;
}

```

Hopefully, that still makes sense. You've gone from having a single context for the lambda expression to one for each iteration of the loop. I'm going to finish this

discussion of captured variables with an even more complicated example, which is a mixture of the two.

CAPTURING VARIABLES FROM MULTIPLE SCOPES

It was the *scope* of the text variable that meant it was instantiated once for each iteration of the loop. But multiple scopes can exist within a single method, and each scope can contain local variable declarations, and a single lambda expression can capture variables from multiple scopes. Listing 3.10 gives an example. You create two delegate instances, each of which captures two variables. They both capture the same `outerCounter` variable, but each captures a separate `innerCounter` variable. The delegates simply print out the current values of the counters and increment them. You execute each delegate twice, which makes the difference between the captured variables clear.

Listing 3.10 Capturing variables from multiple scopes

```
static List<Action> CreateCountingActions()
{
    List<Action> actions = new List<Action>();
    int outerCounter = 0;
    for (int i = 0; i < 2; i++)
    {
        int innerCounter = 0;
        Action action = () =>
        {
            Console.WriteLine(
                "Outer: {0}; Inner: {1}",
                outerCounter, innerCounter);
            outerCounter++;
            innerCounter++;
        };
        actions.Add(action);
    }
    return actions;
}
```

In other code

```
List<Action> actions = CreateCountingActions();
actions[0]();
actions[0]();
actions[1]();
actions[1]();
```

The output of listing 3.10 is as follows:

```
Outer: 0; Inner: 0
Outer: 1; Inner: 1
Outer: 2; Inner: 0
Outer: 3; Inner: 1
```

The first two lines are printed by the first delegate. The last two lines are printed by the second delegate. As I described before the listing, the same outer counter is used by both delegates, but they have independent inner counters.

What does the compiler do with this? Each delegate needs its own context, but that context needs to also refer to a shared context. The compiler creates two private nested classes instead of one. The following listing shows an example of how the compiler could treat listing 3.10.

Listing 3.11 Capturing variables from multiple scopes leads to multiple classes

```

private class OuterContext
{
    public int outerCounter;
}

private class InnerContext
{
    public OuterContext outerContext;
    public int innerCounter;
}

public void Method()
{
    Console.WriteLine(
        "Outer: {0}; Inner: {1}",
        outerContext.outerCounter, innerCounter);
    outerContext.outerCounter++;
    innerCounter++;
}

static List<Action> CreateCountingActions()
{
    List<Action> actions = new List<Action>();
    OuterContext outerContext = new OuterContext();
    outerContext.outerCounter = 0;
    for (int i = 0; i < 2; i++)
    {
        InnerContext innerContext = new InnerContext();
        innerContext.outerContext = outerContext;
        innerContext.innerCounter = 0;
        Action action = innerContext.Method;
        actions.Add(action);
    }
    return actions;
}

```

Context for the outer scope

Context for the inner scope with reference to outer context

Method used to create delegate

Creates a single outer context

Creates an inner context per loop iteration

You'll rarely need to look at the generated code like this, but it can make a difference in terms of performance. If you use a lambda expression in a performance-critical piece of code, you should be aware of how many objects will be created to support the variables it captures.

I could give even more examples with multiple lambda expressions in the same scope capturing different sets of variables or lambda expressions in methods of value types. I find it fascinating to explore compiler-generated code, but you probably wouldn't want a whole book of it. If you ever find yourself wondering how the compiler treats a particular lambda expression, it's easy enough to run a decompiler or ildasm over the result.

So far, you've looked only at converting lambda expressions to delegates, which you could already do with anonymous methods. Lambda expressions have another superpower, however: they can be converted to expression trees.

3.5.3 Expression trees

Expression trees are representations of code as data. This is the heart of how LINQ is able to work efficiently with data providers such as SQL databases. The code you write in C# can be analyzed at execution time and converted into SQL.

Whereas delegates provide code you can run, expression trees provide code you can inspect, a little like reflection. Although you *can* build up expression trees directly in code, it's more common to ask the compiler to do this for you by converting a lambda expression into an expression tree. The following listing gives a trivial example of this by creating an expression tree just to add two numbers together.

Listing 3.12 A simple expression tree to add two integers

```
Expression<Func<int, int, int>> adder = (x, y) => x + y;  
Console.WriteLine(adder);
```

Considering it's only two lines of code, there is a lot going on. Let's start with the output. If you try to print out a regular delegate, the result will be just the type with no indication of the behavior. The output of listing 3.12 shows exactly what the expression tree does, though:

```
(x, y) => x + y
```

The compiler isn't cheating by hardcoding a string somewhere. That string representation is constructed from the expression tree. This demonstrates that the code is available for examination at execution time, which is the whole point of expression trees.

Let's look at the type of adder: `Expression<Func<int, int, int>>`. It's simplest to split it into two parts: `Expression<TDelegate>` and `Func<int, int, int>`. The second part is used as a type argument to the first. The second part is a delegate type with two integer parameters and an integer return type. (The return type is expressed by the last type parameter, so a `Func<string, double, int>` would accept a `string` and a `double` as inputs and return an `int`.)

`Expression<TDelegate>` is the expression tree type associated with `TDelegate`, which must be a delegate type. (That's not expressed as a type constraint, but it's enforced at execution time.) This is only one of the many types involved in expression

trees. They're all in the `System.Linq.Expressions` namespace. The nongeneric `Expression` class is the abstract base class for all the other expression types, and it's also used as a convenient container for factory methods to create instances of the concrete subclasses.

Our adder variable type is an expression tree representation of a function accepting two integers and returning an integer. You then use a lambda expression to assign a value to that variable. The compiler generates code to build the appropriate expression tree at execution time. In this case, it's reasonably simple. You can write the same code yourself, as shown in the following listing.

Listing 3.13 Handwritten code to create an expression tree to add two integers

```
ParameterExpression xParameter = Expression.Parameter(typeof(int), "x");
ParameterExpression yParameter = Expression.Parameter(typeof(int), "y");
Expression body = Expression.Add(xParameter, yParameter);
ParameterExpression[] parameters = new[] { xParameter, yParameter };

Expression<Func<int, int, int>> adder =
    Expression.Lambda<Func<int, int, int>>(body, parameters);
Console.WriteLine(adder);
```

This is a small example, and it's still significantly more long-winded than the lambda expression. By the time you add method calls, property accesses, object initializers, and so on, it gets complex and error prone. That's why it's so important that the compiler can do the work for you by converting lambda expressions into expression trees. There are a few rules around this, though.

LIMITATIONS OF CONVERSIONS TO EXPRESSION TREES

The most important restriction is that only *expression-bodied* lambda expressions can be converted to expression trees. Although our earlier lambda expression of `(x, y) => x + y` was fine, the following code would cause a compilation error:

```
Expression<Func<int, int, int>> adder = (x, y) => { return x + y; };
```

The expression tree API has expanded since .NET 3.5 to include blocks and other constructs, but the C# compiler still has this restriction, and it's consistent with the use of expression trees for LINQ. This is one reason that object and collection initializers are so important: they allow initialization to be captured in a single expression, which means it can be used in an expression tree.

Additionally, the lambda expression can't use the assignment operator, or use C# 4's dynamic typing, or use C# 5's asynchrony. (Although object and collection initializers do use the `=` symbol, that's not the assignment operator in that context.)

COMPILING EXPRESSION TREES TO DELEGATES

The ability to execute queries against remote data sources, as I referred to earlier, isn't the only use for expression trees. They can be a powerful way of constructing efficient delegates dynamically at execution time, although this is typically an area where at

least part of the expression tree is built with handwritten code rather than converted from a lambda expression.

`Expression<TDelegate>` has a `Compile()` method that returns the delegate type. You can then handle this delegate as you do any other. As a trivial example, the following listing takes our earlier adder expression tree, compiles that to a delegate, and then invokes it, producing an output of 5.

Listing 3.14 Compiling an expression tree to a delegate and invoking the result

```
Expression<Func<int, int, int>> adder = (x, y) => x + y;
Func<int, int, int> executableAdder = adder.Compile();
Console.WriteLine(executableAdder(2, 3));
```



Invokes the delegate
as normal

Compiles the
expression tree
to a delegate

This approach can be used in conjunction with reflection for property access and method invocation to produce delegates and then cache them. The result is as efficient as if you'd written the equivalent code by hand. For a single method call or property access, there are already methods to create delegates directly, but sometimes you need additional conversion or manipulation steps, which are easily represented in expression trees.

We'll come back to why expression trees are so important in LINQ when we tie everything together. You have only two more language features to look at. Extension methods come next.

3.6 Extension methods

Extension methods sound pointless when they're first described. They're static methods that can be called as if they're instance methods, based on their first parameter. Suppose you have a static method call like this:

```
ExampleClass.Method(x, y);
```

If you turn `ExampleClass.Method` into an extension method, you can call it like this instead:

```
x.Method(y);
```

That's all extension methods do. It's one of the simplest transformations the C# compiler does. It makes all the difference in terms of code readability when it comes to chaining method calls together, however. You'll look at that later, finally using real examples from LINQ, but first let's look at the syntax.

3.6.1 Declaring an extension method

Extension methods are declared by adding the keyword `this` before the first parameter. The method must be declared in a non-nested, nongeneric static class, and until C# 7.2, the first parameter can't be a `ref` parameter. (You'll see more about that in

section 13.5.) Although the class containing the method can't be generic, the extension method itself can be.

The type of the first parameter is sometimes called the *target* of the extension method and sometimes called the *extended type*. (The specification doesn't give this concept a name, unfortunately.)

As an example from Noda Time, we have an extension method to convert from `DateTimeOffset` to `Instant`. There's already a static method within the `Instant` struct to do this, but it's useful to have as an extension method, too. Listing 3.15 shows the code for the method. For once, I've included the namespace declaration, as that's going to be important when you see how the C# compiler finds extension methods.

Listing 3.15 ToInstant extension method targeting DateTimeOffset from Noda Time

```
using System;

namespace NodaTime.Extensions
{
    public static class DateTimeOffsetExtensions
    {
        public static Instant ToInstant(this DateTimeOffset dateOffset)
        {
            return Instant.FromDateTimeOffset(dateOffset);
        }
    }
}
```

The compiler adds the `[Extension]` attribute to both the method and the class declaring it, and that's all. This attribute is in the `System.Runtime.CompilerServices` namespace. It's a marker indicating the intent that a developer should be able to call `ToInstant()` as if it were declared as an instance method in `DateTimeOffset`.

3.6.2 Invoking an extension method

You've already seen the syntax to invoke an extension method: you call it as if it were an instance method on the type of the first parameter. But you need to make sure that the compiler can find the method as well.

First, there's a matter of priority: if there's a regular instance method that's valid for the method invocation, the compiler will always prefer that over an extension method. It doesn't matter whether the extension method has "better" parameters; if the compiler can use an instance method, it won't even look for extension methods.

After it has exhausted its search for instance methods, the compiler will look for extension methods based on the namespace the calling code is in and any `using` directives present. Suppose you're making a call from the `ExtensionMethodInvocation` class in the `CSharpInDepth.Chapter03` namespace.³ The following

³ If you're following along with the downloaded code, you may have noticed that the samples are in namespaces of `Chapter01`, `Chapter02`, and so on, for simplicity. I've made an exception here for the sake of showing the hierarchical nature of the namespace checks.

listing shows how to do that, giving the compiler all the information it needs to find the extension method.

Listing 3.16 Invoking the `ToInstant()` extension method outside Noda Time

```
using NodaTime.Extensions;
using System;

namespace CSharpInDepth.Chapter03
{
    class ExtensionMethodInvocation
    {
        static void Main()
        {
            var currentInstant =
                DateTimeOffset.UtcNow.ToInstant();
            Console.WriteLine(currentInstant);
        }
    }
}
```

Imports the
NodaTime.Extensions
namespace

Calls the
extension
method

The compiler will check for extension methods in the following:

- Static classes in the `CSharpInDepth.Chapter03` namespace.
- Static classes in the `CSharpInDepth` namespace.
- Static classes in the global namespace.
- Static classes in namespaces specified with `using` namespace directives. (Those are the `using` directives that just specify a namespace, like `using System`.)
- In C# 6 only, static classes specified with `using static` directives. We'll come back to that in section 10.1.

The compiler effectively works its way outward from the deepest namespace out toward the global namespace and looks at each step for static classes either in that namespace or provided by classes made available by `using` directives in the namespace declaration. The details of the ordering are almost never important. If you find yourself in a situation where moving a `using` directive changes which extension method is used, it's probably best to rename one of them. But it's important to understand that within each step, multiple extension methods can be found that would be valid for the call. In that situation, the compiler performs normal overload resolution between all the extension methods it found in that step. After the compiler has located the right method to invoke, the IL it generates for the call is exactly the same as if you'd written a regular static method call instead of using its capabilities as an extension method.

Extension methods can be called on null values

Extension methods differ from instance methods in terms of their null handling. Let's look back at our initial example:

```
x.Method(y);
```

(continued)

If `Method` were an instance method and `x` were a null reference, that would throw a `NullReferenceException`. Instead, if `Method` is an extension method, it'll be called with `x` as the first argument even if `x` is null. Sometimes the method will specify that the first argument must not be null, in which case it should validate it and throw an `ArgumentNullException`. In other cases, the extension method may have been explicitly designed to handle a null first argument gracefully.

Let's get back to why extension methods are important to LINQ. It's time for our first query.

3.6.3 Chaining method calls

Listing 3.17 shows a simple query. It takes a sequence of words, filters them by length, orders them in the natural way, and then converts them to uppercase. It uses lambda expressions and extension methods but no other C# 3 features. We'll put everything else together at the end of the chapter. For the moment, I want to focus on the readability of this simple code.

Listing 3.17 A simple query on strings

```
string[] words = { "keys", "coat", "laptop", "bottle" };
IEnumerable<string> query = words
    .Where(word => word.Length > 4)           ← A simple
    .OrderBy(word => word)                      data source
    .Select(word => word.ToUpper());            Filters, orders,
                                                transforms

foreach (string word in query)
{
    Console.WriteLine(word);                  Displays the results
}
```

Notice the ordering of the `Where`, `OrderBy`, and `Select` calls in our code. That's the order in which the operations happen. The lazy and streaming-where-possible nature of LINQ makes it complicated to talk about exactly what happens when, but the query reads in the same order as it executes. The following listing is the same query but without taking advantage of the fact that these methods are extension methods.

Listing 3.18 A simple query without using extension methods

```
string[] words = { "keys", "coat", "laptop", "bottle" };
IEnumerable<string> query =
    Enumerable.Select(
        Enumerable.OrderBy(
            Enumerable.Where(words, word => word.Length > 4),
            word => word),
        word => word.ToUpper());
```

I've formatted listing 3.18 as readable as I can, but it's still awful. The calls are laid out in the opposite order in the source code to how they'll execute: `Where` is the first thing to execute but the last method call in the listing. Next, it's not obvious which lambda expression goes with which call: `word => word.ToUpper()` is part of the `Select` call, but a huge amount of code is between those two pieces of text.

You can tackle this in another way by assigning the result of each method call to a local variable and then making the method call via that. Listing 3.19 shows one option for doing this. (In this case, you could've just declared the query to start with and reassigned it on each line, but that wouldn't always be the case.) This time, I've also used `var`, just for brevity.

Listing 3.19 A simple query in multiple statements

```
string[] words = { "keys", "coat", "laptop", "bottle" };
var tmp1 = Enumerable.Where(words, word => word.Length > 4);
var tmp2 = Enumerable.OrderBy(tmp1, word => word);
var query = Enumerable.Select(tmp2, word => word.ToUpper());
```

This is better than listing 3.18; the operations are back in the right order, and it's obvious which lambda expression is used for which operation. But the extra local variable declarations are a distraction, and it's easy to end up using the wrong one.

The benefits of method chaining aren't limited to LINQ, of course. Using the result of one call as the starting point of another call is common. But extension methods allow you to do this in a readable way for any type, rather than the type itself declaring the methods that support chaining. `IEnumerable<T>` doesn't know anything about LINQ; its sole responsibility is to represent a general sequence. It's the `System.Linq.Enumerable` class that adds all the operations for filtering, grouping, joining, and so on.

C# 3 could've stopped here. The features described so far would already have added a lot of power to the language and enabled many LINQ queries to be written in a perfectly readable form. But when queries get more complex, particularly when they include joins and groupings, using the extension methods directly can get complicated. Enter query expressions.

3.7 Query expressions

Although almost all features in C# 3 contribute to LINQ, only *query expressions* are specific to LINQ. Query expressions allow you to write concise code by using query-specific clauses (`select`, `where`, `let`, `group by`, and so on). The query is then translated into a nonquery form by the compiler and compiled as normal.⁴ Let's start with a brief example to make this clearer. As a reminder, in listing 3.17 you had this query:

```
IEnumerable<string> query = words
    .Where(word => word.Length > 4)
    .OrderBy(word => word)
    .Select(word => word.ToUpper());
```

⁴ This sounds like macros in C, but it's a little more involved than that. C# still doesn't have macros.

The following listing shows the same query written as a query expression.

Listing 3.20 Introductory query expression with filtering, ordering, and projection

```
IEnumerable<string> query = from word in words
                            where word.Length > 4
                            orderby word
                            select word.ToUpper();
```

The section of listing 3.20 in bold is the query expression, and it's very concise indeed. The repetitive use of `word` as a parameter to lambda expressions has been replaced by specifying the name of a *range variable* once in the `from` clause, and then using it in each of the other clauses. What happens to the query expression in listing 3.20?

3.7.1 Query expressions translate from C# to C#

In this book, I've expressed many language features in terms of more C# source code. For example, when looking at captured variables in section 3.5.2, I showed C# code that you could've written to achieve the same result as using a lambda expression. That's just for the purpose of explaining the code generated by the compiler. I wouldn't expect the compiler to generate any C#. The specification describes the effects of capturing variables rather than a source code translation.

Query expressions work differently. The specification describes them as a syntactic translation that occurs before any overload resolution or binding. The code in listing 3.20 doesn't just have the same eventual effect as listing 3.17; it's really translated into the code in listing 3.17 before further processing. The language has no specific expectation about what the result of that further processing will be. In many cases, the result of the translation will be calls to extension methods, but that's not required by the language specification. They could be instance method calls or invocations of delegates returned by properties named `Select`, `Where`, and so on.

The specification of query expressions puts in place an expectation of certain methods being available, but there's no specific requirement for them all to be present. For example, if you write an API with suitable `Select`, `OrderBy`, and `Where` methods, you could use the kind of query shown in listing 3.20 even though you couldn't use a query expression that includes a `join` clause.

Although we're not going to look at every clause available in query expressions in detail, I need to draw your attention to two related concepts. In part, these provide greater justification for the language designers introducing query expressions into the language.

3.7.2 Range variables and transparent identifiers

Query expressions introduce *range variables*, which aren't like any other regular variables. They act as the per item input within each clause of the query. You've already seen how the `from` clause at the start of a query expression introduces a range

variable. Here's the query expression from listing 3.20 again with the range variable highlighted:

```
from word in words
where word.Length > 4
orderby word
select word.ToUpper()
```

The diagram shows the range variable 'word' starting from the 'from' clause and pointing to its use in the 'select' clause. A callout box labeled 'Introduces range variable in a from clause' points to the 'from' clause, and another callout box labeled 'Uses the range variable in the following clauses' points to the 'select' clause.

That's simple to understand when there's only one range variable, but that initial `from` clause isn't the only way a range variable can be introduced. The simplest example of a clause that introduces a new range variable is probably `let`. Suppose you want to refer to the length of the word multiple times in your query without having to call the `Length` property every time. For example, you could `orderby` it and include it in the output. The `let` clause allows you to write the query as shown in the following listing.

Listing 3.21 A `let` clause introducing a new range variable

```
from word in words
let length = word.Length
where length > 4
orderby length
select string.Format("{0}: {1}", length, word.ToUpper());
```

You now have two range variables in scope at the same time, as you can see from the use of both `length` and `word` in the `select` clause. That raises the question of how this can be represented in the query translation. You need a way of taking our original sequence of words and creating a sequence of word/length pairs, effectively. Then within the clauses that can use those range variables, you need to access the relevant item within the pair. The following listing shows how listing 3.21 is translated by the compiler using an anonymous type to represent the pair of values.

Listing 3.22 Query translation using a transparent identifier

```
words.Select(word => new { word, length = word.Length })
    .Where(tmp => tmp.length > 4)
    .OrderBy(tmp => tmp.length)
    .Select(tmp =>
        string.Format("{0}: {1}", tmp.length, tmp.word.ToUpper()));
```

The name `tmp` here isn't part of the query translation. The specification uses `*` instead, and there's no indication of what name should be given to the parameter when building an expression tree representation of the query. The name doesn't matter because you don't see it when you write the query. This is called a *transparent identifier*.

I'm not going into all the details of query translation. That could be a whole chapter on its own. But I wanted to bring up transparent identifiers for two reasons. First, if you're aware of how extra range variables are introduced, you won't be surprised when you see them if you ever decompile a query expression. Second, they provide the biggest motivation for using query expressions, in my experience.

3.7.3 Deciding when to use which syntax for LINQ

Query expressions can be appealing, but they're not always the simplest way of representing a query. They always require a `from` clause to start with and either a `select` or `group by` clause to end with. That sounds reasonable, but it means that if you want a query that performs a single filtering operation, for example, you end up with quite a lot of baggage. For example, if you take just the filtering part of our word-based query, you'd have the following query expression:

```
from word in words  
where word.Length > 4  
select word
```

Compare that with the method syntax version of the query:

```
words.Where(word => word.Length > 4)
```

They both compile to the same code,⁵ but I'd use the second syntax for such a simple query.

NOTE There's no single ubiquitous term for not using query expression syntax. I've seen it called *method syntax*, *dot syntax*, *fluent syntax*, and *lambda syntax*, to name just four. I'll call it *method syntax* consistently, but if you hear other terms for it, don't try to look for a subtle difference in meaning.

Even when the query gets a little more complicated, method syntax can be more flexible. Many methods are available within LINQ that have no corresponding query expression syntax, including overloads of `Select` and `Where` that present the index of the item within the sequence as well as the item itself. Additionally, if you want a method call at the end of the query (for example, `ToList()` to materialize the result as a `List<T>`), you have to put the whole query expression in parentheses, whereas with method syntax you add the call on the end.

I'm not as down on query expressions as that may sound. In many cases, there's no clear winner between the two syntax options, and I'd probably include our earlier filter, order, project example in that set. Query expressions really shine when the compiler is doing more work for you by handling all those transparent identifiers. You can do it all by hand, of course, but I've found that building up anonymous types as results and deconstructing them in each subsequent step gets annoying quickly. Query expressions make all of that much easier.

The upshot of all of this is that I strongly recommend that you become comfortable in both styles of query. If you tie yourself to always using query expressions or never using query expressions, you'll be missing out on opportunities to make your code more readable. We've covered all the features in C# 3, but I'm going to take a moment to step back and show how they fit together to form LINQ.

⁵ The compiler has special handling for `select` clauses that select just the current query item.

3.8 The end result: LINQ

I'm not going to attempt to cover the various LINQ providers available these days. The LINQ technology I use most (by far) is LINQ to Objects, using the `Enumerable` static class and delegates. But in order to show how all the pieces come into play, let's imagine that you have a query from something like Entity Framework. This isn't real code that you can test, but it would be fine if you had a suitable database structure:

```
var products = from product in dbContext.Products
               where product.StockCount > 0
               orderby product.Price descending
               select new { product.Name, product.Price };
```

In this single example of a mere four lines, all of these features are used:

- Anonymous types, including projection initializers (to select just the name and price of the product)
- Implicit typing using `var`, because otherwise you couldn't declare the type of the `products` variable in a useful way
- Query expressions, which you could do without in this case, but which make life a lot simpler for more-complicated queries
- Lambda expressions, which are the result of the query expression translation
- Extension methods, which allow the translated query to be expressed via the `Queryable` class because of `dbContext.Products` implementing `IQueryable<Product>`
- Expression trees, which allow the logic in the query to be passed to the LINQ provider as data, so it can be converted into SQL and executed efficiently at the database

Take away any one of these features, and LINQ would be significantly less useful. Sure, you could have in-memory collection processing without expression trees. You could write readable simple queries without query expressions. You could have dedicated classes with all the relevant methods without using extension methods. But it all fits together beautifully.

Summary

- All the features in C# 3 are related to working with data in some form or other, and most are critical parts of LINQ.
- Automatically implemented properties provide a concise way of exposing state that doesn't need any extra behavior.
- Implicit typing with the `var` keyword (and for arrays) is necessary for working with anonymous types but also convenient to avoid long-winded repetition.
- Object and collection initializers make initialization simpler and more readable. They also allow initialization to occur as a single expression, which is crucial for working with other aspects of LINQ.

- Anonymous types allow you to effectively create a type just for a single local purpose in a lightweight way.
- Lambda expressions provide an even simpler way of constructing delegates than anonymous methods. They also allow code to be expressed as data via expression trees, which can be used by LINQ providers to convert C# queries into other forms such as SQL.
- Extension methods are static methods that can be called as if they were instance methods elsewhere. This allows for fluent interfaces to be written even for types that weren't originally designed that way.
- Query expressions are translated into more C# that uses lambda expressions to express the query. Although these are great for complex queries, simpler ones are often easier to write using method syntax.

C# 4: Improving interoperability

This chapter covers

- Using dynamic typing for interoperability and simpler reflection
- Providing default values for parameters so the caller doesn't need to specify them
- Specifying names for arguments to make calls clearer
- Coding against COM libraries in a more streamlined fashion
- Converting between generic types with generic variance

C# 4 was an interesting release. The most dramatic change was the introduction of dynamic typing with the `dynamic` type. This feature makes C# statically typed (for most code) and dynamically typed (when using `dynamic`) in the same language. That's rare within programming languages.

Dynamic typing was introduced for interoperability, but that's turned out not to be relevant in many developers' day-to-day work. The major features in other

releases (generics, LINQ, async/await) have become a natural part of most C# developers' toolkits, but dynamic typing is still used relatively rarely. I'm sure it's useful to those who need it, and at the very least it's an interesting feature.

The other features in C# 4 also improve interoperability, particularly with COM. Some improvements are specific to COM, such as named indexers, implicit ref arguments, and embedded interop types. Optional parameters and named arguments are useful with COM, but they can also be used in purely managed code. These two are the features from C# 4 that I use on a daily basis.

Finally, C# 4 exposes a feature of generics that was present in the CLR from v2 (the first runtime version that included generics). Generic variance is simultaneously simple and complex. At first glance, it sounds obvious: a sequence of strings is obviously a sequence of objects, for example. But then we discover that a list of strings isn't a list of objects, dashing the expectations of some developers. It's a useful feature, but one that's prone to inducing headaches when you examine it closely. Most of the time, you can take advantage of it without even being aware that you're doing so. Hopefully, the coverage in this chapter will mean that if you do need to look closer because your code isn't working as you expect, you'll be in a good position to fix the problem without getting confused. We'll start off by looking at dynamic typing.

4.1 Dynamic typing

Some features come with a lot of new syntax, but after you've explained the syntax, there's not much left to say. Dynamic typing is the exact opposite: the syntax is extremely simple, but I could go into almost endless detail about the impact and implementation. This section shows you the basics and then goes into some of the details before closing with a few suggestions about how and when to use dynamic typing.

4.1.1 Introduction to dynamic typing

Let's start with an example. The following listing shows two attempts to take a substring from some text. At the moment, I'm not trying to explain why you'd want to use dynamic typing, just what it does.

Listing 4.1 Taking a substring by using dynamic typing

```
dynamic text = "hello world";
string world = text.Substring(6);           ← Declares a variable
Console.WriteLine(world);                  with the dynamic type

string broken = text.SUBSTR(6);            ← Calls the Substring
Console.WriteLine(broken);                method; this works.

Tries to call SUBSTR; this
throws an exception.
```

A lot is going on here for such a small amount of code. The most important aspect is that it compiles at all. If you changed the first line to declare `text` by using the `string` type, the call to `SUBSTR` would fail at compile time. Instead, the compiler is happy to compile it without even looking for a method called `SUBSTR`. It doesn't look for `Substring` either. Instead, both lookups are performed at execution time.

At execution time, the second line will look for a method called `Substring` that can be called with an argument of 6. That method is found and returns a string, which you then assign to the `world` variable and print in a regular way. When the code looks for a method called `SUBSTR` that can be called with an argument of 6, it doesn't find any such method, and the code fails with a `RuntimeBinderException`.

As mentioned in chapter 3, this process of looking up the meaning of a name in a certain context is called *binding*. Dynamic typing is all about changing when binding happens from compile time to execution time. Instead of just generating IL that calls a method with a precise signature determined at execution time, the compiler generates IL that performs the binding and then acts on the result. All of this is triggered by using the `dynamic` type.

WHAT IS THE DYNAMIC TYPE?

Listing 4.1 declared the `text` variable as being of type `dynamic`:

```
dynamic text = "hello world";
```

What is the `dynamic` type? It's different from other types you see in C#, because it exists only as far as the C# language is concerned. There's no `System.Type` associated with it, and the CLR doesn't know about it at all. Anytime you use `dynamic` in C#, the IL uses `object` decorated with `[Dynamic]` if necessary.

NOTE If the `dynamic` type is used in a method signature, the compiler needs to make that information available for code compiling against it. There's no need to do this for local variables.

The basic rules of the `dynamic` type are simple:

- 1 There's an implicit conversion from any nonpointer type to `dynamic`.
- 2 There's an implicit conversion from an expression of type `dynamic` to any non-pointer type.
- 3 Expressions that involve a value of type `dynamic` are usually bound at execution time.
- 4 Most expressions that involve a value of type `dynamic` have a compile-time type of `dynamic` as well.

You'll look at the exceptions to the last two points shortly. Using this list of rules, you can look at listing 4.1 again with fresh eyes. Let's consider the first two lines:

```
dynamic text = "hello world";
string world = text.Substring(6);
```

In the first line, you're converting from `string` to `dynamic`, which is fine because of rule 1. The second line demonstrates all three of the other rules:

- `text.Substring(6)` is bound at execution time (rule 3).
- The compile-time type of that expression is `dynamic` (rule 4).
- There's an implicit conversion from that expression to `string` (rule 2).

The conversion from an expression of type `dynamic` to a nondynamic type is dynamically bound, too. If you declared the `world` variable to be of type `int`, that would compile but fail at execution time with a `RuntimeBinderException`. If you declared it to be of type `XNamespace`, that would compile and then at execution time the binder would use the user-defined implicit conversion from `string` to `XNamespace`. With this in mind, let's look at more examples of dynamic binding.

APPLYING DYNAMIC BINDING IN A VARIETY OF CONTEXTS

So far, you've seen dynamic binding based on the dynamic target of a method call and then a conversion, but almost any aspect of execution can be dynamic. The following listing demonstrates this in the context of the addition operator and performs three kinds of addition based on the type of the dynamic value at execution time.

Listing 4.2 Addition of dynamic values

```
static void Add(dynamic d)
{
    Console.WriteLine(d + d);           ← Performs addition based on
}                                         the type at execution time

Add("text");
Add(10);
Add(TimeSpan.FromMinutes(45));          | Calls the method
                                         with different values
```

The results of listing 4.2 are as follows:

```
texttext
20
01:30:00
```

Each kind of addition makes sense for the type involved, but in a statically typed context, they'd look different. As one final example, the following listing shows how method overloading behaves with dynamic method arguments.

Listing 4.3 Dynamic method overload resolution

```
static void SampleMethod(int value)
{
    Console.WriteLine("Method with int parameter");
}

static void SampleMethod(decimal value)
{
    Console.WriteLine("Method with decimal parameter");
}

static void SampleMethod(object value)
{
    Console.WriteLine("Method with object parameter");
}
```

```

static void CallMethod(dynamic d)
{
    SampleMethod(d);
}

CallMethod(10);
CallMethod(10.5m);
CallMethod(10L);
CallMethod("text");

```

← Calls SampleMethod dynamically

Indirectly calls SampleMethod with different types

The output of listing 4.3 is as follows:

```

Method with int parameter
Method with decimal parameter
Method with decimal parameter
Method with object parameter

```

The third and fourth lines of the output are particularly interesting. They show that the overload resolution at execution time is still aware of conversions. In the third line, a long value is converted to decimal rather than int, despite being an integer in the range of int. In the fourth line, a string value is converted to object. The aim is that, as far as possible, the binding at execution time should behave the same way it would've at compile time, just using the types of the dynamic values as they're discovered at execution time.

Only dynamic values are considered dynamically

The compiler works hard to make sure the right information is available at execution time. When binding involves multiple values, the compile-time type is used for any values that are statically typed, but the execution-time type is used for any values of type dynamic. Most of the time, this nuance is irrelevant, but I've provided an example with comments in the downloadable source code.

The result of any dynamically bound method call has a compile-time type of dynamic. When binding occurs, if the chosen method has a void return type and the result of the method was used (for example, being assigned to a variable), then binding fails. That's the case for most dynamically bound operations: the compiler has little information about what the dynamic operation will entail. That rule has a few exceptions.

WHAT CAN THE COMPILER CHECK IN DYNAMICALLY BOUND CONTEXTS?

If the context of a method call is known at compile time, the compiler is able to check what methods exist with the specified name. If no methods could possibly match at execution time, a compile-time error is still reported. This applies to the following:

- Instance methods and indexers where the target isn't a dynamic value
- Static methods
- Constructors

The following listing shows various examples of calls using dynamic values that fail at compile time.

Listing 4.4 Examples of compile-time failures involving dynamic values

```

dynamic d = new object();
int invalid1 = "text".Substring(0, 1, 2, d);           ← No String.Substring method
                                                        with four parameters
bool invalid2 = string.Equals<int>("foo", d);        ← No generic
                                                        String.Equals
                                                        method
string invalid3 = new string(d, "broken");            ← No String constructors with
                                                        two parameters accepting a
                                                        string as a second argument
char invalid4 = "text"[d, d];                         ← No String indexer
                                                        with two parameters

```

Just because the compiler is able to tell that these particular examples are definitely broken doesn't mean it'll always be able to do so. Dynamic binding is always a bit of a leap into the unknown unless you're very careful about the values involved.

The examples I've given would still use dynamic binding if they compiled. There are only a few cases where that's not the case.

WHAT OPERATIONS INVOLVING DYNAMIC VALUES AREN'T DYNAMICALLY BOUND?

Almost everything you do with a dynamic value involves binding of some kind and finding the right method call, property, conversion, operator, and so on. There are just a few things that the compiler doesn't need to generate any binding code for:

- Assignments to a variable of type `object` or `dynamic`. No conversion is required, so the compiler can just copy the existing reference.
- Passing an argument to a method with a corresponding parameter of type `object` or `dynamic`. That's like assigning a variable, but the variable is the parameter.
- Testing a value's type with the `is` operator.
- Attempting to convert a value with the `as` operator.

Although the execution-time binding infrastructure is happy to find user-defined conversions if you convert a dynamic value to a specific type with a cast or just do so implicitly, the `is` and `as` operators never use user-defined conversions, so no binding is required. In a similar way, *almost* all operations with dynamic values have a result that is also dynamic.

WHAT OPERATIONS INVOLVING DYNAMIC VALUES STILL HAVE A STATIC TYPE?

Again, the compiler wants to help as much as it can. If an expression can always be of only one specific type, the compiler is happy to make that the compile-time type of the expression. For example, if `d` is a variable of type `dynamic`, the following are true:

- The expression `new SomeType(d)` has a compile-time type of `SomeType`, even though the constructor is bound dynamically at execution time.

- The expression `d is SomeType` has a compile-time type of `bool`.
- The expression `d as SomeType` has a compile-time type of `SomeType`.

That's all the detail you need for this introduction. In section 4.1.4, you'll look at unexpected twists, both at compile time and execution time. But now that you have the flavor of dynamic typing, you can look at some of its power beyond performing regular binding at execution time.

4.1.2 Dynamic behavior beyond reflection

One use for dynamic typing is to effectively ask the compiler and framework to perform reflection operations for you based on the members declared in types in the usual way. Although that's a perfectly reasonable use, dynamic typing is more extensible. Part of the reason for its introduction was to allow better interoperability with dynamic languages that allow on-the-fly changes in binding. Many dynamic languages allow interception of calls at execution time. This has usages such as transparent caching and logging or making it look like there are functions and fields that are never declared by name in the source code.

IMAGINARY EXAMPLE OF DATABASE ACCESS

As an (unimplemented) example of the kind of thing you might want to do, imagine you have a database containing a table of books, including their authors. Dynamic typing would make this sort of code possible:

```
dynamic database = new Database(connectionString);
var books = database.Books.SearchByAuthor("Holly Webb");
foreach (var book in books)
{
    Console.WriteLine(book.Title);
}
```

This would involve the following dynamic operations:

- The `Database` class would respond to a request for the `Books` property by querying the database schema for a table called `Books` and returning some sort of table object.
- That table object would respond to the `SearchByAuthor` method call by spotting that it started with `SearchBy` and looking for a column called `Author` within the schema. It would then generate SQL to query by that column using the provided argument and return a list of row objects.
- Each row object would respond to the `Title` property by returning the value of the `Title` column.

If you're used to Entity Framework or a similar object-relational mapping (ORM), this may not sound like anything new. You can write classes fairly easily that enable the same kind of querying code or generate those classes from the schema. The difference here is that it's all dynamic: there's no `Book` or `BooksTable` class. It all just happens at

execution time. In section 4.1.5, I'll talk about whether that's a good or a bad thing in general, but I hope you can at least see how it could be useful in some situations.

Before I introduce you to the types that allow all of this to happen, let's look at two examples that *are* implemented. First, you'll look at a type in the framework, and then at Json.NET.

EXPANDOOBJECT: A DYNAMIC BAG OF DATA AND METHODS

The .NET Framework provides a type called `ExpandoObject` in the namespace `System.Dynamic`. It operates in two modes depending on whether you're using it as a dynamic value. The following listing gives a brief example to help you make sense of the description that follows it.

Listing 4.5 Storing and retrieving items in an ExpandoObject

```
dynamic expando = new ExpandoObject();
expando.SomeData = "Some data";           | Assigns data
Action<string> action =                  | to a property
    input => Console.WriteLine("The input was '{0}'", input);
expando.FakeMethod = action;              | Assigns a delegate
                                           | to a property

Console.WriteLine(expando.SomeData);        | Accesses the data and
expando.FakeMethod("hello");              | delegate dynamically

IDictionary<string, object> dictionary = expando;
Console.WriteLine("Keys: {0}",
    string.Join(", ", dictionary.Keys));   | Treats the ExpandoObject as a
                                           | dictionary to print the keys

dictionary["OtherData"] = "other";
Console.WriteLine(expando.OtherData);      | Populates data with the static context
                                           | and fetches it from the dynamic value
```

When `ExpandoObject` is used in a statically typed context, it's a dictionary of name/value pairs, and it implements `IDictionary<string, object>` as you'd expect from a normal dictionary. You can use it that way, looking up keys that are provided at execution time and so on.

More important, it also implements `IDynamicMetaObjectProvider`. This is the entry point for dynamic behavior. You'll look at the interface itself later, but `ExpandoObject` implements it so you can access the dictionary keys by name within code. When you invoke a method on an `ExpandoObject` in a dynamic context, it'll look up the method name as a key in the dictionary. If the value associated with that key is a delegate with appropriate parameters, the delegate is executed, and the result of the delegate is used as the result of the method call.

Listing 4.5 stored only one data value and one delegate, but you can store many with whatever names you want. It's just a dictionary that can be accessed dynamically.

You could implement much of the earlier database example by using `ExpandoObject`. You'd create one to represent the `Books` table and then represent each book with a separate `ExpandoObject`, too. The table would have a key of `SearchByAuthor` with a suitable delegate value to execute the query. Each book would have

a key of `Title` storing the title and so on. In practice, though, you'd want to implement `IDynamicMetaObjectProvider` directly or use `DynamicObject`. Before diving into those types, let's take a look at another implementation: accessing JSON data dynamically.

THE DYNAMIC VIEW OF JSON.NET

JSON is everywhere these days, and one of the most popular libraries for consuming and creating JSON is `Json.NET`.¹ It provides multiple ways of handling JSON, including parsing straight to user-provided classes and parsing to an object model that's closer to LINQ to XML. The latter is called *LINQ to JSON* with types such as `JObject`, `JArray`, and `JProperty`. It can be used like LINQ to XML, with access via strings, or it can be used dynamically. The following listing shows both approaches for the same JSON.

Listing 4.6 Using JSON data dynamically

```
string json = @"  
{  
    'name': 'Jon Skeet',  
    'address': {  
        'town': 'Reading',  
        'country': 'UK'  
    }  
}"".Replace('\'', '\"');  
  
JObject obj1 = JObject.Parse(json); ← Parses the JSON  
Console.WriteLine(obj1["address"]["town"]); ← to a JObject  
  
dynamic obj2 = obj1; ← Uses the statically  
Console.WriteLine(obj2.address.town); ← typed view  
  
Uses the dynamically  
typed view
```

The diagram illustrates the flow of JSON data. It starts with a **Hardcoded sample JSON** string. This string is passed to **Parses the JSON to a JObject**, which creates a `JObject` instance named `obj1`. This `obj1` is then used in a `Console.WriteLine` statement to output the value of the `address.town` property. Finally, the `obj1` variable is converted to a `dynamic` object named `obj2`, and its `address.town` property is accessed again, demonstrating the dynamic nature of the view.

This JSON is simple but includes a nested object. The second half of the code shows how that can be accessed by either using the indexers within LINQ to JSON or using the dynamic view it provides.

Which of these do you prefer? Arguments exist for and against each approach. Both are prone to typos, whether within a string literal or the dynamic property access. The statically typed view lends itself to extracting the property names into constants for reuse, but the dynamically typed view is simpler to read when prototyping. I'll make some suggestions for when and where dynamic typing is appropriate in section 4.1.5, but it's worth reflecting on your initial reactions before you get there. Next we'll take a quick look at how to do all of this yourself.

IMPLEMENTING DYNAMIC BEHAVIOR IN YOUR OWN CODE

Dynamic behavior is complicated. Let's get that out of the way to start with. Please don't expect to come away from this section ready to write a production-ready optimized implementation of whatever amazing idea you have. This is only a starting point. That

¹ Other JSON libraries are available, of course. I just happen to be most familiar with `Json.NET`.

said, it should be enough to let you explore and experiment so you can decide how much effort you wish to invest in learning all the details.

When I presented `ExpandoObject`, I mentioned that it implements the interface `IDynamicMetaObjectProvider`. This is the interface signifying that an object implements its own dynamic behavior instead of just being happy to let the reflection-based infrastructure work in the normal way. As an interface, it looks deceptively simple:

```
public interface IDynamicMetaObjectProvider
{
    DynamicMetaObject GetMetaObject(Expression parameter);
}
```

The complexity lies in `DynamicMetaObject`, which is the class that drives everything else. Its official documentation gives a clue as to the level you need to think at when working with it:

Represents the dynamic binding and a binding logic of an object participating in the dynamic binding.

Even having used the class, I wouldn't like to claim I fully understand that sentence, nor could I write a better description. Typically, you'd create a class deriving from `DynamicMetaObject` and override some of the virtual methods it provides. For example, if you want to handle method invocations dynamically, you'd override this method:

```
public virtual DynamicMetaObject BindInvokeMember
    (InvokeMemberBinder binder, DynamicMetaObject[] args);
```

The `binder` parameter gives information such as the name of the method being called and whether the caller expects binding to be performed case sensitively. The `args` parameter provides the arguments provided by the caller in the form of more `DynamicMetaObject` values. The result is yet another `DynamicMetaObject` representing how the method call should be handled. It doesn't perform the call immediately but creates an expression tree representing what the call would do.

All of this is extremely complicated but allows for complex situations to be handled efficiently. Fortunately, you don't have to implement `IDynamicMetaObjectProvider` yourself, and I'm not going to try to do so. Instead, I'll give an example using a much friendlier type: `DynamicObject`.

The `DynamicObject` class acts as a base class for types that want to implement dynamic behavior as simply as possible. The result may not be as efficient as directly implementing `IDynamicMetaObjectProvider` yourself, but it's much easier to understand.

As a simple example, you're going to create a class (`SimpleDynamicExample`) with the following dynamic behavior:

- Invoking any method on it prints a message to the console, including the method name and arguments.
- Fetching a property usually returns that property name with a prefix to show you really called into the dynamic behavior.

The following listing shows how you would use the class.

Listing 4.7 Example of intended use of dynamic behavior

```
dynamic example = new SimpleDynamicExample();
example.CallSomeMethod("x", 10);
Console.WriteLine(example.SomeProperty);
```

The output should be as follows:

```
Invoked: CallSomeMethod(x, 10)
Fetched: SomeProperty
```

There's nothing special about the names `CallSomeMethod` and `SomeProperty`, but you could've reacted to specific names in different ways if you'd wanted to. Even the simple behavior described so far would be tricky to get right using the low-level interface, but the following listing shows how easy it is with `DynamicObject`.

Listing 4.8 Implementing SimpleDynamicExample

```
class SimpleDynamicExample : DynamicObject
{
    public override bool TryInvokeMember(
        InvokeMemberBinder binder,
        object[] args,
        out object result)
    {
        Console.WriteLine("Invoked: {0}({1})",
            binder.Name, string.Join(", ", args));
        result = null;
        return true;
    }

    public override bool TryGetMember(
        GetMemberBinder binder,
        out object result)
    {
        result = "Fetched: " + binder.Name;
        return true;
    }
}
```

Handles
method
calls

Handles
property
access

As with the methods on `DynamicMetaObject`, you still receive binders when overriding the methods in `DynamicObject`, but you don't need to worry about expression trees or other `DynamicMetaObject` values anymore. The return value from each method indicates whether the dynamic object successfully handled the operation. If you return `false`, a `RuntimeBinderException` will be thrown.

That's all I'm going to show you in terms of implementing dynamic behavior, but I hope the simplicity of listing 4.8 will encourage you to experiment with `DynamicObject`. Even if you never use it in production, playing with it can be a lot of fun. If

you want to give it a try but don't have concrete ideas, you could always try implementing the Database example I gave at the start of this section. As a reminder, here's the code you'd be trying to enable:

```
dynamic database = new Database(connectionString);
var books = database.Books.SearchByAuthor("Holly Webb");
foreach (var book in books)
{
    Console.WriteLine(book.Title);
}
```

Next, you'll take a look at the code the C# compiler generates when it encounters dynamic values.

4.1.3 A brief look behind the scenes

You're probably aware by now that I enjoy looking at the IL that the C# compiler uses to implement its various features. You've already looked at how captured variables in lambda expressions can result in extra classes being generated and how lambda expressions converted to expression trees result in calls to methods in the `Expression` class. Dynamic typing works a little bit like expression trees in terms of creating a data representation of the source code, but on a larger scale.

This section goes into even less detail than the previous one. Although the details are interesting, you almost certainly won't need to know them.² The good news is that it's all open source, so you can go as low-level as you want to if you find yourself tantalized by this brief introduction to the topic. We'll start off by considering which subsystem is responsible for what aspect of dynamic typing.

WHO DOES WHAT?

Normally when you consider a C# feature, it's natural to divide responsibility into three areas:

- The C# compiler
- The CLR
- The framework libraries

Some features are purely in the domain of the C# compiler. Implicit typing is an example of this. The framework doesn't need to provide any types to support `var`, and the runtime is blissfully unaware of whether you used implicit or explicit typing.

At the other end of the spectrum is generics, which require significant compiler support, runtime support, and framework support in terms of the reflection APIs. LINQ is somewhere in between: the compiler provides the various features you saw in chapter 3, and the framework provides not only the implementation of LINQ to Objects but also the API for expression trees. On the other hand, the runtime didn't

² And to be honest, I don't know enough details to do the whole topic justice.

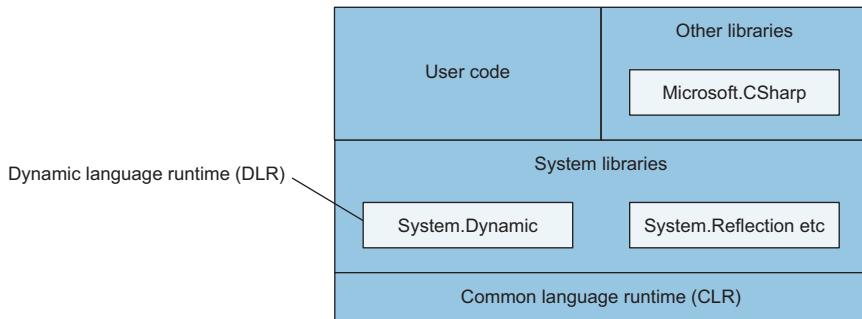


Figure 4.1 Graphical representation of components involved in dynamic typing

need to change. For dynamic typing, the picture is a little more complicated. Figure 4.1 gives a graphical representation of the elements involved.

The CLR didn't require changes, although I believe there were optimizations from v2 to v4 that were somewhat driven by this work. The compiler is obviously involved in generating different IL, and we'll look at an example of this in a moment. For framework/library support, there are two aspects. The first is the *Dynamic Language Runtime* (DLR), which provides language-agnostic infrastructure such as `DynamicMetaObject`. That's responsible for executing all the dynamic behavior. But a second library isn't part of the core framework itself: `Microsoft.CSharp.dll`.

NOTE This library ships with the framework but isn't part of the system framework libraries as such. I find it helpful to think of it as if it were a third-party dependency, where the third party happens to be Microsoft. On the other hand, the Microsoft C# compiler is fairly tightly coupled to it. It doesn't fit into any box particularly neatly.

This library is responsible for anything C# specific. For example, if you make a method call in which one argument is a dynamic value, it's this library that performs the overload resolution at execution time. It's a copy of the part of the C# compiler responsible for binding, but it does so in the context of all the dynamic APIs.

If you've ever seen a reference to `Microsoft.CSharp.dll` in your project and wondered what it was for, that's the reason. If you don't use dynamic typing anywhere, you can safely remove the reference. If you do use dynamic typing but remove the reference, you'll get a compile-time error as the C# compiler generates calls into that assembly. Speaking of code generated by the C# compiler, let's have a look at some now.

THE IL GENERATED FOR DYNAMIC TYPING

We're going to go right back to our initial example of dynamic typing but make it even shorter. Here are the first two lines of dynamic code I showed you:

```
dynamic text = "hello world";
string world = text.Substring(6);
```

Pretty simple, right? There are two dynamic operations here:

- The call to the `Substring` method
- The conversion from the result to a string

The following listing is a decompiled version of the code generated from those two lines. I've included the surrounding context of a class declaration and `Main` method just for clarity.

Listing 4.9 The result of decompiling two simple dynamic operations

```
using Microsoft.CSharp.RuntimeBinder;
using System;
using System.Runtime.CompilerServices;

class DynamicTypingDecompiled
{
    private static class CallSites           ← Cache of
    {
        public static CallSite<Func<CallSite, object, int, object>>
            method;
        public static CallSite<Func<CallSite, object, string>>
            conversion;
    }

    static void Main()
    {
        object text = "hello world";           ← Creates a call site for the
        if (CallSites.method == null)           ← method call if necessary
        {
            CSharpArgumentInfo[] argumentInfo = new[]
            {
                CSharpArgumentInfo.Create(
                    CSharpArgumentInfoFlags.None, null),
                CSharpArgumentInfo.Create(
                    CSharpArgumentInfoFlags.Constant |
                    CSharpArgumentInfoFlags.UseCompileTimeType,
                    null)
            };
            CallSiteBinder binder =
                Binder.InvokeMember(CSharpBinderFlags.None, "Substring",
                    null, typeof(DynamicTypingDecompiled), argumentInfo);
            CallSites.method =
                CallSite<Func<CallSite, object, int, object>>.Create(binder);
        }

        if (CallSites.conversion == null)         ← Creates a call site for the
        {                                       ← conversion if necessary
            CallSiteBinder binder =
                Binder.Convert(CSharpBinderFlags.None, typeof(string),
                    typeof(DynamicTypingDecompiled));
            CallSites.conversion =
                CallSite<Func<CallSite, object, string>>.Create(binder);
        }
    }
}
```

```

object result = CallSites.method.Target(
    CallSites.method, text, 6);

string str =
    CallSites.conversion.Target(CallSites.conversion, result);
}

}

```

I apologize for this formatting. I've done what I can to make it readable, but it's a lot of code that involves a lot of long names. The good news is, you're almost certain to never need to look at code like this except for the sake of interest. One point to note is that `CallSite` is in the `System.Runtime.CompilerServices` namespace as it's language neutral, whereas the `Binder` class being used is from `Microsoft.CSharp.RuntimeBinder`.

As you can tell, a lot of *call sites* are involved. Each call site is cached by the generated code, and multiple levels of caching are within the DLR as well. Binding is a reasonably involved process. The cache within the call site improves performance by storing the result of each binding operation to avoid redundant work while being aware that the same call could end up with different binding results if some of the context changes between calls.

The result of all this effort is a system that's remarkably efficient. It doesn't perform quite as well as statically typed code, but it's surprisingly close. I expect that in most cases where dynamic typing is an appropriate choice for other reasons, its performance won't be a limiting factor. To wrap up the coverage of dynamic typing, I'll explain a few limitations you may encounter and then give a little guidance around when and how dynamic typing is an effective choice.

4.1.4 Limitations and surprises in dynamic typing

Integrating dynamic typing into a language that was designed from the start to be statically typed is difficult. It's no surprise that in a few places, the two don't play nicely together. I've put together a list of some of the aspects of dynamic typing that include limitations or potential surprises to encounter at execution time. The list isn't exhaustive, but it covers the most commonly seen problems.

THE DYNAMIC TYPE AND GENERICS

Using the `dynamic` type with generics can be interesting. Rules are applied at compile time about where you can use `dynamic`:

- A type can't specify that it implements an interface using `dynamic` anywhere in a type argument.
- You can't use `dynamic` anywhere in type constraints.
- A class can specify a base class that uses `dynamic` in a type argument, even as part of an interface type argument.
- You can use `dynamic` as an interface type argument for variables.

Here are some examples of invalid code:

```
class DynamicSequence : IEnumerable<dynamic>
class DynamicListSequence : IEnumerable<List<dynamic>>
class DynamicConstraint1<T> : IEnumerable<T> where T : dynamic
class DynamicConstraint2<T> : IEnumerable<T> where T : List<dynamic>
```

But all of these are valid:

```
class DynamicList : List<dynamic>
class ListOfDynamicSequences : List<IEnumerable<dynamic>>
IEnumerable<dynamic> x = new List<dynamic> { 1, 0.5 }.Select(x => x * 2);
```

EXTENSION METHODS

The execution-time binder doesn't resolve extension methods. It could conceivably do so, but it'd need to keep additional information about every relevant `using` directive at every method call site. It's important to note that this doesn't affect statically bound calls that happen to use a dynamic type somewhere within a type argument. So, for example, the following listing compiles and runs with no problems.

Listing 4.10 A LINQ query over a list of dynamic values

```
List<dynamic> source = new List<dynamic>
{
    5,
    2.75,
    TimeSpan.FromSeconds(45)
};
IEnumerable<dynamic> query = source.Select(x => x * 2);
foreach (dynamic value in query)
{
    Console.WriteLine(value);
}
```

The only dynamic operations here are the multiplication (`x * 2`) and the overload resolution in `Console.WriteLine`. The call to `Select` is bound as normal at compile time. As an example of what will fail, let's try making the source itself dynamic and simplify the LINQ operation you're using to `Any()`. (If you kept using `Select` as before, you'd run into another problem that you'll look at in a moment.) The following listing shows the changes.

Listing 4.11 Attempting to call an extension method on a dynamic target

```
dynamic source = new List<dynamic>
{
    5,
    2.75,
    TimeSpan.FromSeconds(45)
};
bool result = source.Any();
```

I haven't included the output part, because execution doesn't reach there. Instead, it fails with a `RuntimeBinderException` because `List<T>` doesn't include a method called `Any`.

If you want to call an extension method as if its target were a dynamic value, you need to do so as a regular static method call. For example, you could rewrite the last line of listing 4.11 to the following:

```
bool result = Enumerable.Any(source);
```

The call will still be bound at execution time, but only in terms of overload resolution.

ANONYMOUS FUNCTIONS

Anonymous functions have three limitations. For the sake of simplicity, I'll show them all with lambda expressions.

First, anonymous methods can't be assigned to a variable of type `dynamic`, because the compiler doesn't know what kind of delegate to create. It's fine if you either cast or use an intermediate statically typed variable (and then copy the value), and you can invoke the delegate dynamically, too. For example, this is invalid:

```
dynamic function = x => x * 2;
Console.WriteLine(function(0.75));
```

But this is fine and prints 1.5:

```
dynamic function = (Func<dynamic, dynamic>) (x => x * 2);
Console.WriteLine(function(0.75));
```

Second, and for the same underlying reason, lambda expressions can't appear within dynamically bound operations. This is the reason I didn't use `Select` in listing 4.11 to demonstrate the problem with extension methods. Here's what listing 4.11 would've looked like otherwise:

```
dynamic source = new List<dynamic>
{
    5,
    2.75,
    TimeSpan.FromSeconds(45)
};
dynamic result = source.Select(x => x * 2);
```

You know that wouldn't work at execution time because it wouldn't be able to find the `Select` extension method, but it doesn't even compile because of the use of the lambda expression. The workaround for the compile-time issue is the same as before: just cast the lambda expression to a delegate type or assign it to a statically typed variable first. That would still fail at execution time for extension methods such as `Select`, but it would be fine if you were calling a regular method such as `List<T>.Find`, for example.

Finally, lambda expressions that are converted to expression trees must not contain any dynamic operations. This may sound slightly odd, given the way the DLR uses expression trees internally, but it's rarely an issue in practice. In most cases where

expression trees are useful, it's unclear what dynamic typing means or how it could possibly be implemented.

As an example, you can attempt to tweak listing 4.10 (with the statically typed source variable) to use `IQueryable<T>`, as shown in the following listing.

Listing 4.12 Attempting to use a dynamic element type in an `IQueryable<T>`

```
List<dynamic> source = new List<dynamic>
{
    5,
    2.75,
    TimeSpan.FromSeconds(45)
};
IEnumerable<dynamic> query = source
    .AsQueryable()
    .Select(x => x * 2);
```

This line now
fails to compile.

The result of the `AsQueryable()` call is an `IQueryable<dynamic>`. This is statically typed, but its `Select` method accepts an expression tree rather than a delegate. That means the lambda expression (`x => x * 2`) would have to be converted to an expression tree, but it's performing a dynamic operation, so it fails to compile.

ANONYMOUS TYPES

I mentioned this issue when I first covered anonymous types, but it bears repeating: anonymous types are generated as regular classes in IL by the C# compiler. They have internal access, so nothing can use them outside the assembly they're declared in. Normally that's not an issue, as each anonymous type is typically used only within a single method. With dynamic typing, you can read properties of instances of anonymous types, but only if that code has access to the generated class. The following listing shows an example of this where it *is* valid.

Listing 4.13 Dynamic access to a property of an anonymous type

```
static void PrintName(dynamic obj)
{
    Console.WriteLine(obj.Name);
}

static void Main()
{
    var x = new { Name = "Abc" };
    var y = new { Name = "Def", Score = 10 };
    PrintName(x);
    PrintName(y);
}
```

This listing has two anonymous types, but the binding process doesn't care whether it's binding against an anonymous type. It does check that it has access to the properties it finds, though. If you split this code across two assemblies, that would cause a

problem; the binder would spot that the anonymous type is internal to the assembly where it's created and throw a `RuntimeBinderException`. If you run into this problem and can use `[InternalsVisibleTo]` to allow the assembly performing the dynamic binding to have access to the assembly where the anonymous type is created, that's a reasonable workaround.

EXPLICIT INTERFACE IMPLEMENTATION

The execution-time binder uses the execution-time type of any dynamic value and then binds in the same way as if you'd written that as the compile-time type of a variable. Unfortunately, that doesn't play nicely with the existing C# feature of explicit interface implementation. When you use explicit interface implementation, that effectively says that the member being implemented is available only when you're using the interface view over the object instead of the type itself.

It's easier to show this than to explain it. The following listing uses `List<T>` as an example.

Listing 4.14 Example of explicit interface implementation

```
List<int> list1 = new List<int>();           ← Compile-time  
Console.WriteLine(list1.IsFixedSize);          error  
  
IList list2 = list1;                         ← Succeeds;  
Console.WriteLine(list2.IsFixedSize);          prints False  
  
dynamic list3 = list1;                        ← Execution-time  
Console.WriteLine(list3.IsFixedSize);          error
```

`List<T>` implements the `IList` interface. The interface has a property called `IsFixedSize`, but the `List<T>` class implements that explicitly. Any attempt to access it via an expression with a static type of `List<T>` will fail at compile time. You can access it via an expression with a static type of `IList`, and it'll always return `false`. But what about accessing it dynamically? The binder will always use the concrete type of the dynamic value, so it fails to find the property, and a `RuntimeBinderException` is thrown. The workaround here is to convert the dynamic value back to the interface (via casting or a separate variable) if you know that you want to use an interface member.

I'm sure that anyone who works with dynamic typing on a regular basis would be able to regale you with a long list of increasingly obscure corner cases, but the preceding items should keep you from being surprised *too* often. We'll complete our coverage of dynamic typing with a little guidance about when and how to use it.

4.1.5 Usage suggestions

I'll be up front about this: I'm generally not a fan of dynamic typing. I can't remember the last time I used it in production code, and I'd do so only warily and after a lot of testing for correctness and performance.

I'm a sucker for static typing. In my experience, it gives four significant benefits:

- When I make mistakes, I'm likely to discover them earlier—at compile time rather than execution time. That's particularly important with code paths that may be hard to test exhaustively.
- Editors can provide code completion. This isn't particularly important in terms of speed of typing, but it's great as a way of exploring what I might want to do next, particularly if I'm using a type I'm unfamiliar with. Editors for dynamic languages can provide remarkable code-completion facilities these days, but they'll never be quite as precise as those for statically typed languages, because there just isn't as much information available.
- It makes me think about the API I'm providing, in terms of parameters, return types, and so on. After I've made decisions about which types to accept and return, that acts as ready-made documentation: I need to add comments only for anything that isn't otherwise obvious, such as the range of acceptable values.
- By doing work at compile time instead of execution time, statically typed code usually has performance benefits over dynamically typed code. I don't want to emphasize this too much, as modern runtimes can do amazing things, but it's certainly worth considering.

I'm sure a dynamic typing aficionado would be able to give you a similar list of awesome benefits of dynamic typing, but I'm not the right person to do so. I suspect those benefits are more readily available in a language designed with dynamic typing right from the start. C# is *mostly* a statically typed language, and its heritage is clear, which is why the corner cases I listed earlier exist. That said, here are a few suggestions about when you might want to use dynamic typing.

SIMPLER REFLECTION

Suppose you find yourself using reflection to access a property or method; you know the name at compile time, but you can't refer to the static type for whatever reason. It's much simpler to use dynamic typing to ask the runtime binder to perform that access than to do it directly with the reflection API. The benefit increases if you'd otherwise need to perform multiple steps of reflection. For example, consider a code snippet like this:

```
dynamic value = ...;
value.SomeProperty.SomeMethod();
```

The reflection steps involved would be as follows:

- 1 Fetch the `PropertyInfo` based on the type of the initial value.
- 2 Fetch the value of that property and remember it.
- 3 Fetch the `MethodInfo` based on the type of the property result.
- 4 Execute the method on the property result.

By the time you've added validation to check that the property and method both exist, you're looking at several lines of code. The result would be no safer than the dynamic approach shown previously, but it would be a lot harder to read.

COMMON MEMBERS WITHOUT A COMMON INTERFACE

Sometimes you do know all the possible types of a value in advance, and you want to use a member with the same name on all of them. If the types implement a common interface or share a common base class that declares the member, that's great, but that doesn't always happen. If each of them declares that member independently (and if you can't change that), you're left with unpleasant choices.

This time, you don't need to use reflection, but you might need to perform several repetitive steps of check the type, cast, access the member. C# 7 patterns make this significantly simpler, but it can still be repetitive. Instead, you can use dynamic typing to effectively say "Trust me, I know this member will be present, even though I can't express it in a statically typed way." I'd be comfortable doing this within tests (where the cost of being wrong is a test failure), but in production code I'd be much more cautious.

USING A LIBRARY BUILT FOR DYNAMIC TYPING

The .NET ecosystem is pretty rich and is getting better all the time. Developers are creating all kinds of interesting libraries, and I suspect some may embrace dynamic typing. For example, I can imagine a library designed to allow for easy prototyping with REST- or RPC-based APIs with no code generation involved. That could be useful in the initial phase of development while everything is quite fluid before generating a statically typed library for later development.

This is similar to the Json.NET example you looked at earlier. You may well want to write classes to represent your data model after that model is well-defined, but when prototyping, it may be simpler to change the JSON and then the code that's accessing it dynamically. Likewise, you'll see later how COM improvements mean that often you can end up working with dynamic typing instead of performing a lot of casting.

In a nutshell, I think it still makes sense to use static typing where it's simple to do so, but you should accept dynamic typing as a potentially useful tool for some situations. I encourage you to weigh the pros and cons in each context. Code that's acceptable for a prototype or even in test code may not be suitable for production code, for example.

Beyond code that you might write for professional purposes, the ability to respond with dynamic behavior by using `DynamicObject` or `IDynamicMetaObjectProvider` certainly gives a lot of scope for fun development. However much I may shy away from dynamic typing myself, it's been well designed and implemented in C# and provides a rich avenue for exploration.

Our next feature is somewhat different, although both will come together when you look at COM interoperability. We're back to static typing and one specific aspect of it: providing arguments for parameters.

4.2 **Optional parameters and named arguments**

Optional parameters and named arguments have a limited scope: given a method, constructor, indexer, or delegate that you want to call, how do you provide the arguments for the call? *Optional parameters* allow the caller to omit an argument entirely, and *named arguments* allow the caller to make it clear to both the compiler and any human reader which parameter an argument is intended to relate to.

Let's start with a simple example and then dive into the details. In this whole section, I'm going to consider only methods. The same rules apply to all the other kinds of members that can have parameters.

4.2.1 Parameters with default values and arguments with names

The following listing shows a simple method with three parameters, two of which are optional. Multiple calls to the method then demonstrate different features.

Listing 4.15 Calling a method with optional parameters

```
static void Method(int x, int y = 5, int z = 10)    ← One required parameter, two optional
{
    Console.WriteLine("x={0}; y={1}; z={2}", x, y, z);    ← Just print the parameter values.
}
...
Method(1, 2, 3);          ← x=1; y=2; z=3
Method(x: 1, y: 2, z: 3); ← x=1; y=2; z=3
Method(z: 3, y: 2, x: 1); ← x=1; y=2; z=3
Method(1, 2);             ← x=1; y=2; z=10
Method(1, y: 2);          ← x=1; y=2; z=10
Method(1, z: 3);          ← x=1; y=5; z=3           ← x=1; y=2; z=10
Method(1);                ← x=1; y=5; z=10
Method(x: 1);              ← x=1; y=5; z=10
```

Figure 4.2 shows the same method declaration and one method call, just to make the terminology clear.

The syntax is simple:

- A parameter can specify a *default value* after its name with an equal sign between the name and the value. Any parameter with a default value is *optional*; any parameter without a default value is *required*. Parameters with `ref` or `out` modifiers aren't permitted to have default values.
- An argument can specify a name before the value with a colon between the name and the value. An argument without a name is called a *positional argument*.

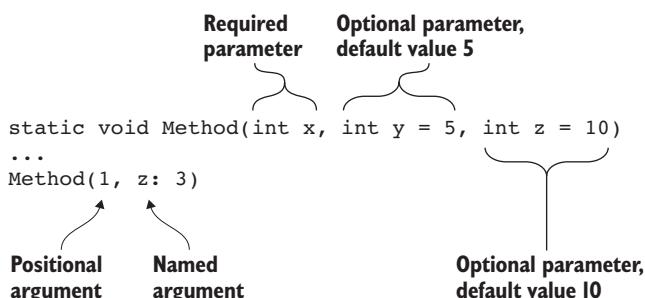


Figure 4.2 Syntax of optional/required parameters and named/positional arguments

The default value for a parameter must be one of the following expressions:

- A compile-time constant, such as a numeric or string literal, or the null literal.
- A default expression, such as `default(CancellationToken)`. As you'll see in section 14.5, C# 7.1 introduces the *default literal*, so you can write `default` instead of `default(CancellationToken)`.
- A new expression, such as `new Guid()` or `new CancellationToken()`. This is valid only for value types.

All optional parameters must come after all required parameters, with an exception for parameter arrays. (Parameter arrays are parameters with the `params` modifier.)

WARNING Even though you can declare a method with an optional parameter followed by a parameter array, it ends up being confusing to call. I urge you to avoid this, and I won't go into how calls to such methods are resolved.

The purpose of making a parameter optional is to allow the caller to omit it if the value it *would supply* is the same as the default value. Let's look at what the compiler handles a method call that can involve default parameters and/or named arguments.

4.2.2 Determining the meaning of a method call

If you read the specification, you'll see that the process of working out which argument corresponds to which parameter is part of overload resolution and is intertwined with type inference. This is more complicated than you might otherwise expect, so I'm going to simplify things here. We'll focus on a single method signature, assume it's the one that has already been chosen by overload resolution, and take it from there.

The rules are reasonably simple to list:

- All positional arguments must come before all named arguments. This rule is relaxed slightly in C# 7.2, as you'll see in section 14.6.
- Positional arguments always correspond to a parameter in the same position in the method signature. The first positional argument corresponds to the first parameter, the second positional argument corresponds to the second parameter, and so on.
- Named arguments match by name instead of position: an argument named `x` corresponds to a parameter named `x`. Named arguments can be specified in any order.
- Any parameter can have only one corresponding argument. You can't specify the same name in two named arguments, and you can't use a named argument for a parameter that already has a corresponding positional argument.
- Every required parameter must have a corresponding argument to provide a value.
- Optional parameters are permitted not to have a corresponding argument, in which case the compiler will supply the default value as an argument.

To see these rules in action, let's consider our original simple method signature:

```
static void Method(int x, int y = 5, int z = 10)
```

You can see that `x` is a required parameter because it doesn't have a default value, but `y` and `z` are optional parameters. Table 4.1 shows several valid calls and their results.

Table 4.1 Examples of valid method calls for named arguments and optional parameters

Call	Resulting arguments	Notes
Method(1, 2, 3)	x=1; y=2; z=3	All positional arguments. Regular call from before C# 4.
Method(1)	x=1; y=5; z=10	Compiler supplies values for <code>y</code> and <code>z</code> , as there are no corresponding arguments.
Method()	n/a	Invalid: no argument corresponds to <code>x</code> .
Method(y: 2)	n/a	Invalid: no argument corresponds to <code>x</code> .
Method(1, z: 3)	x=1; y=5; z=3	Compiler supplies value for <code>y</code> as there's no corresponding argument. It was skipped by using a named argument for <code>z</code> .
Method(1, x: 2, z: 3)	n/a	Invalid: two arguments correspond to <code>x</code> .
Method(1, y: 2, y: 2)	n/a	Invalid: two arguments correspond to <code>y</code> .
Method(z: 3, y: 2, x: 1)	x=1; y=2; z=3	Named arguments can be in any order,

There are two more important aspects to note when it comes to evaluating method calls. First, arguments are evaluated in the order they appear in the source code for the method call, left to right. In most cases, this wouldn't matter, but if argument evaluation has side effects, it can. As an example, consider these two calls to our sample method:

```
int tmp1 = 0;
Method(x: tmp1++, y: tmp1++, z: tmp1++);           ← x=0; y=1; z=2

int tmp2 = 0;
Method(z: tmp2++, y: tmp2++, x: tmp2++);           ← x=2; y=1; z=0
```

The two calls differ only in terms of the order of their named arguments, but that affects the values that are passed into the method. In both cases, the code is harder to read than it might be. When side effects of argument evaluation are important, I encourage you to evaluate them as separate statements and assign to new local variables that are then passed directly to the method as arguments, like this:

```
int tmp3 = 0;
int argX = tmp3++;
int argY = tmp3++;
int argZ = tmp3++;
Method(x: argX, y: argY, z: argZ);
```

At this point, whether you name the arguments doesn't change the behavior; you can choose whichever form you find most readable. The separation of argument evaluation from method invocation makes the order of argument evaluation simpler to understand, in my opinion.

The second point to note is that if the compiler has to specify any default values for parameters, those values are embedded in the IL for the calling code. There's no way for the compiler to say "I don't have a value for this parameter; please use whatever default you have." That's why the default values have to be compile-time constants, and it's one of the ways in which optional parameters affect versioning.

4.2.3 Impact on versioning

Versioning of public APIs in libraries is a hard problem. It's really hard and significantly less clear-cut than we like to pretend. Although semantic versioning says that any breaking change means you need to move to a new major version, pretty much any change can break some code that depends on the library, if you're willing to include obscure cases. That said, optional parameters and named arguments are particularly tricky for versioning. Let's have a look at the various factors.

PARAMETER NAME CHANGES ARE BREAKING

Suppose you have a library containing the method that you previously looked at, but it's public:

```
public static Method(int x, int y = 5, int z = 10)
```

Now suppose you want to change that to the following in a new version:

```
public static Method(int a, int b = 5, int c = 10)
```

That's a breaking change; any code that uses named arguments when calling the method will be broken, as the names they specified before no longer exist. Check your parameter names as carefully as you check your type and member names!

DEFAULT VALUE CHANGES ARE AT LEAST SURPRISING

As I've noted, default values are compiled into the IL of the calling code. When that's within the same assembly, changing the default value doesn't cause a problem. When it's in a different assembly, a change to the default value will be visible only when the calling code is recompiled.

That's not always a problem, and if you anticipate that you might want to change the default value, it's not entirely unreasonable to state that explicitly in the method documentation. But it could definitely surprise some developers using your code, particularly if complicated dependency chains are involved. One way of avoiding this is to use a dedicated default value that always means "Let the method choose at execution time." For example, if you have a method that would normally have an `int` parameter, you could use `Nullable<int>` instead, with a default value of `null` meaning "the method will choose." You can change the implementation of the method later to

make a different choice, and every caller using the new version will get the new behavior, whether they've recompiled or not.

ADDING OVERLOADS IS FIDDLY

If you thought overload resolution was tricky in a single-version scenario, it becomes a lot worse when you're trying to add an overload without breaking anyone. All original method signatures must be present in the new version to avoid breaking binary compatibility, and all calls against the original methods should either resolve to the same calls, or at least *equivalent* calls, in the new version. Whether a parameter is required or optional isn't part of the method signature itself; you don't break binary compatibility by changing an optional parameter to be required, or vice versa. But you might break source compatibility. If you're not careful, you can easily introduce ambiguity in overload resolution by adding a new method with more optional parameters.

If two methods are both applicable within overload resolution (both make sense with respect to the call) and neither is better than the other in terms of the argument-to-parameter conversions involved, then default parameters can be used as a tiebreak. A method that has no optional parameters without corresponding arguments is "better" than a method with at least one optional parameter without a corresponding argument. But a method with one unfilled parameter is no better than a method with two such parameters.

If you can possibly get away without adding overloads to methods when optional parameters are involved, I strongly advise that you do so—and, ideally, bear that in mind from the start. One pattern to consider for methods that might have a lot of options is to create a class representing all those options and then take that as an optional parameter in method calls. You can then add new options by adding properties to the options class without changing the method signature at all.

Despite all these caveats, I'm still in favor of optional parameters when they make sense to simplify calling code for common cases, and I'm a big fan of the ability to name arguments to clarify calling code. This is particularly relevant when multiple parameters of the same type could be confused with each other. As one example, I always use them when I need to call the Windows Forms `MessageBox.Show` method. I can never remember whether the title of the message box or the text comes first. IntelliSense can help me when I'm writing the code, but it's not as obvious when I'm reading it, unless I use named arguments:

```
MessageBox.Show(text: "This is text", caption: "This is the title");
```

Our next topic is one that many readers may have no need for and other readers will use every day. Although COM is a legacy technology in many contexts, a huge amount of code still uses it.

4.3

COM interoperability improvements

Before C# 4, VB was simply a better language to use if you wanted to interoperate with COM components. It's always been a somewhat more relaxed language, at least if you ask it to be, and it has had named arguments and optional parameters from the start.

C# 4 makes life much simpler for those working with COM. That said, if you're not using COM, you won't miss out on anything important by skipping this section. None of the features I go into here is relevant outside COM.

NOTE COM is the Component Object Model introduced by Microsoft in 1993 as a cross-language form of interoperability on Windows. A full description is beyond the scope of this book, but you're likely to know about it if you need to know about it. The most commonly used COM libraries are probably those for Microsoft Office.

Let's start with a feature that goes beyond the language. It's mostly about deployment, although it also impacts how the operations are exposed.

4.3.1 Linking primary interop assemblies

When you code against a COM type, you use an assembly generated for the component library. Usually, you use a *primary interop assembly* (PIA) generated by the component publisher. You can use the Type Library Importer tool (`tlbimp`) to generate this for your own COM libraries.

Before C# 4, the complete PIA had to be present on the machine where the code finally ran, and it had to be the same version as the one that you compiled against. This either meant shipping the PIA along with your application or trusting that the right version would already be installed.

From C# 4 and Visual Studio 2010 onward, you can choose to *link* the PIA instead of *referencing* it. In Visual Studio, in the property page for the reference, this is the Embed Interop Types option.

When this option is set to True, the relevant parts of the PIA are embedded directly into your assembly. Only the bits you use within your application are included. When the code runs, it doesn't matter whether the exact same version of the component you used to compile against is present on the client machine, so long as it has everything that your application needs. Figure 4.3 shows the difference between referencing (the old way) and linking (the new way) in terms of how the code runs.

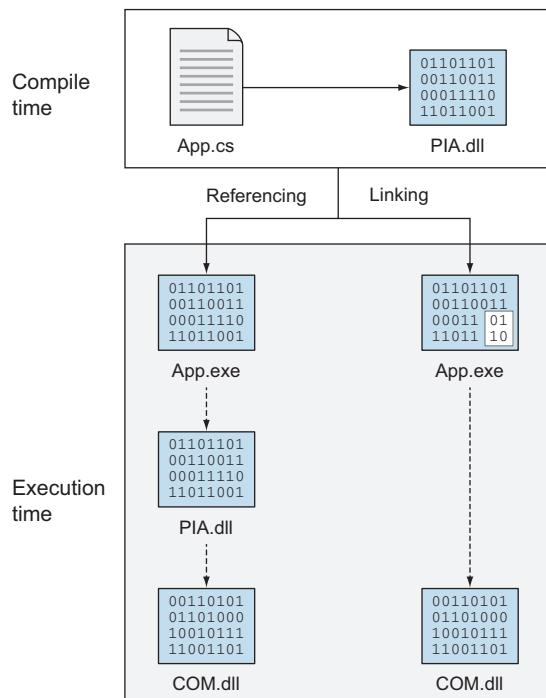


Figure 4.3 Comparing referencing and linking

In addition to deployment changes, linking the PIA affects how the VARIANT type is treated within the COM type. When the PIA is referenced, any operations returning a VARIANT value would be exposed using the object type in C#. You'd then have to cast that to the appropriate type to use its methods and properties.

When the PIA is linked instead, dynamic is returned instead of object. As you saw earlier, there's an implicit conversion from an expression of type dynamic to any nonpointer type, which is then checked at execution time. The following listing shows an example of opening Excel and populating 20 cells in a range.

Listing 4.16 Setting a range of values in Excel with implicit dynamic conversion

```
var app = new Application { Visible = true };
app.Workbooks.Add();
Worksheet sheet = app.ActiveSheet;
Range start = sheet.Cells[1, 1];
Range end = sheet.Cells[1, 20];
sheet.Range[start, end].Value = Enumerable.Range(1, 20).ToArray();
```

Listing 4.16 silently uses some of the features coming up later, but for the moment focus on the assignments to sheet, start, and end. Each would need a cast normally, as the value being assigned would be of type object. You don't have to specify the static types for the variables; if you used var or dynamic for the variable types, you'd be using dynamic typing for more operations. I prefer to specify the static type where I know what I expect it to be, partly for the implicit validation this performs and partly to enable IntelliSense in the code that follows.

For COM libraries that use VARIANT extensively, this is one of the most important benefits of dynamic typing. The next COM feature also builds on a new feature in C# 4 and takes optional parameters to a new level.

4.3.2 Optional parameters in COM

Some COM methods have a *lot* of parameters, and often they're all ref parameters. This meant that prior to C# 4, a simple act like saving a file in Word could be extremely painful.

Listing 4.17 Creating a Word document and saving it before C# 4

```
object missing = Type.Missing;           ↪ Placeholder variable  
                                         ↪ for ref parameters
Application app = new Application { Visible = true };    ↪ Starts Word
Document doc = app.Documents.Add
    ref missing, ref missing,
    ref missing, ref missing);
Paragraph para = doc.Paragraphs.Add(ref missing);
para.Range.Text = "Awkward old code";                      ↪ Creates and populates  
                                         ↪ a document
```

```

object fileName = "demo1.docx";
doc.SaveAs2(ref fileName, ref missing,
    ref missing, ref missing, ref missing,
    ref missing, ref missing, ref missing,
    ref missing, ref missing, ref missing,
    ref missing, ref missing);

doc.Close(ref missing, ref missing, ref missing);
app.Application.Quit(
    ref missing, ref missing);

```

Saves the document

Closes Word

A lot of code is required just to create and save a document, including 20 occurrences of `ref missing`. It's hard to see the useful part of the code within the forest of arguments you don't care about.

C# 4 provides features that all work together to make this much simpler:

- Named arguments can be used to make it clear which argument should correspond to which parameter, as you've already seen.
- Just for COM libraries, values can be specified directly as arguments for `ref` parameters. The compiler will create a local variable behind the scenes and pass that by reference.
- Just for COM libraries, `ref` parameters can be optional and then omitted in the calling code. `Type.Missing` is used as the default value.

With all of these features in play, you can transform listing 4.17 into much shorter and cleaner code.

Listing 4.18 Creating a Word document and saving it using C# 4

```

Application app = new Application { Visible = true };
Document doc = app.Documents.Add();
Paragraph para = doc.Paragraphs.Add();
para.Range.Text = "Simple new code";                                ← Optional parameters omitted everywhere

doc.SaveAs2(FileName: "demo2.docx");                                ← Named argument used for clarity
doc.Close();
app.Application.Quit();

```

This is a dramatic transformation in readability. All 20 occurrences of `ref missing` are gone, as is the variable itself. As it happens, the argument you pass to `SaveAs2` corresponds to the first parameter of the method. You could use a positional argument instead of a named argument, but specifying the name adds clarity. If you also wanted to specify a value for a later parameter, you could do so by name without providing values for all the other parameters in between.

That argument to `SaveAs2` also demonstrates the implicit `ref` feature. Instead of having to declare a variable within an initial value of `demo2.docx` and then pass that by reference, you can pass the value directly, as far as our source code is concerned.

The compiler handles turning it into a `ref` parameter for you. The final COM-related feature exposes another aspect where VB is slightly richer than C#.

4.3.3 Named indexers

Indexers have been present in C# since the beginning. They're primarily used for collections: retrieving an element from a list by index or retrieving a value from a dictionary by key, for example. But C# indexers are never named in source code. You can write only the *default indexer* for the type. You can specify a name by using an attribute, and that name will be consumed by other languages, but C# doesn't let you differentiate between indexers by name. At least, it didn't until C#4.

Other languages allow you to write and consume indexers with names, so you can access different aspects of an object via indexes using the name to make it clear what you want. C# still doesn't do this for regular .NET code, but it makes an exception just for COM types. An example will make this clearer.

The `Application` type in Word exposes a named indexer called `SynonymInfo`. It's declared like this:

```
SynonymInfo SynonymInfo[string Word, ref object LanguageId = Type.Missing]
```

Prior to C# 4, you could call the indexer as if it were a method called `get_SynonymInfo`, but that's somewhat awkward. In C# 4, you can access it by name, as shown in the following listing.

Listing 4.19 Accessing a named indexer

```
Application app = new Application { Visible = false };

object missing = Type.Missing;
SynonymInfo info = app.get_SynonymInfo("method", ref missing);           | Accessing synonyms
Console.WriteLine("'method' has {0} meanings", info.MeaningCount);        | prior to C# 4

info = app.SynonymInfo["index"];                                         <----- Simpler code using
Console.WriteLine("'index' has {0} meanings", info.MeaningCount);          | a named indexer
```

Listing 4.19 shows how optional parameters can be used in named indexers as well as regular method calls. The code for before C# 4 has to declare a variable and pass it by reference to the awkwardly named method. With C# 4, you can use the indexer by name, and you can omit the argument for the second parameter.

That was a brief run through the COM-related features in C# 4, but I hope the benefits are obvious. Even though I don't work with COM regularly, the changes shown here would make me a lot less despondent if I ever need to in the future. The extent of the benefit will depend on how the COM library you're working with is structured. For example, if it uses a lot of `ref` parameters and VARIANT return types, the difference will

be more significant than a library with few parameters and concrete return types. But even just the option of linking the PIA could make deployment significantly simpler.

We're coming toward the end of C# 4 now. The final feature can be a bit tricky to get your head around, but it's also one you may use without even thinking about it.

4.4 Generic variance

Generic variance is easier to show than to describe. It's about safely converting between generic types based on their type arguments and paying particular attention to the direction in which data travels.

4.4.1 Simple examples of variance in action

We'll start with an example using a familiar interface, `IEnumerable<T>`, which represents a sequence of elements of type `T`. It makes sense that any sequence of strings is also a sequence of objects, and variance allows that:

```
IEnumerable<string> strings = new List<string> { "a", "b", "c" };
IEnumerable<object> objects = strings;
```

That may seem so natural that you'd be surprised if it failed to compile, but that's exactly what would've happened before C# 4.

NOTE I'm using `string` and `object` consistently in these examples because they're classes that all C# developers know about and aren't tied to any particular context. Other classes with the same base class/derived class relationship would work just as well.

There are potentially more surprises to come; not everything that sounds like it should work does work, even with C# 4. For example, you might try to extend the reasoning about sequences to lists. Is any list of strings a list of objects? You might think so, but it's not:

```
IList<string> strings = new List<string> { "a", "b", "c" };  
IList<object> objects = strings;
```

← Invalid: no conversion from
IList<string> to IList<object>

What's the difference between `IEnumerable<T>` and `IList<T>`? Why isn't this allowed? The answer is that it wouldn't be safe, because the methods within `IList<T>` allow values of type `T` as inputs as well as outputs. Every way you can use an `IEnumerable<T>` ends up with `T` values being returned as output, but `IList<T>` has methods like `Add` that accept a `T` value as input. That would make it dangerous to allow variance. You can see this if you try to extend our invalid example a little:

```
IList<string> strings = new List<string> { "a", "b", "c" };
IList<object> objects = strings;
objects.Add(new object());
string element = strings[3];
```

← Adds an object to the list

← Retrieves it as a string

Every line other than the second one makes sense on its own. It's fine to add an object reference to an `IList<object>`, and it's fine to take a string reference from an `IList<string>`. But if you can treat a list of strings as a list of objects, those two abilities come into conflict. The language rules that make the second line invalid are effectively protecting the rest of the code.

So far, you've seen values being returned as output (`IEnumerable<T>`) and values being used as both input and output (`IList<T>`). In some APIs, values are always used only as input. The simplest example of this is the `Action<T>` delegate, where you pass in a value of type `T` when you invoke the delegate. Variance still applies here, but in the opposite direction. This can be confusing to start with.

If you have an `Action<object>` delegate, that can accept any object reference. It can definitely accept a string reference, and the language rules allow you to convert from `Action<object>` to `Action<string>`:

```
Action<object> objectAction = obj => Console.WriteLine(obj);
Action<string> stringAction = objectAction;
stringAction("Print me");
```

With those examples in hand, I can define some terminology:

- *Covariance* occurs when values are returned only as output.
- *Contravariance* occurs when values are accepted only as input.
- *Invariance* occurs when values are used as input and output.

Those definitions are deliberately slightly vague for now. They're more about the general concepts than they are about C#. We can tighten them up after you've looked at the syntax C# uses to specify variance.

4.4.2 Syntax for variance in interface and delegate declarations

The first thing to know about variance in C# is that it can be specified only for interfaces and delegates. You can't make a class or struct covariant, for example. Next, variance is defined separately for each type parameter. Although you might loosely say "`IEnumerable<T>` is covariant," it would be more precise to say "`IEnumerable<T>` is covariant *in T*." That then leads to syntax for interface and delegate declarations in which each type parameter has a separate modifier. Here are the declarations for the `IEnumerable<T>` and `IList<T>` interfaces and `Action<T>` delegate:

```
public interface IEnumerable<out T>
public delegate void Action<in T>
public interface IList<T>
```

As you can see, the modifiers `in` and `out` are used to specify the variance of a type parameter:

- A type parameter with the `out` modifier is covariant.
- A type parameter with the `in` modifier is contravariant.
- A type parameter with no modifiers is invariant.

The compiler checks that the modifier you've used is suitable given the rest of the declaration. For example, this delegate declaration is invalid because a covariant type parameter is used as input:

```
public delegate void InvalidCovariant<out T>(T input)
```

And this interface declaration is invalid because a contravariant type parameter is used as output:

```
public interface IInvalidContravariant<in T>
{
    T GetValue();
}
```

Any single type parameter can have only one of these modifiers, but two type parameters in the same declaration can have different modifiers. For example, consider the `Func<T, TResult>` delegate. That accepts a value of type `T` and returns a value of type `TResult`. It's natural for `T` to be contravariant and `TResult` to be covariant. The delegate declaration is as follows:

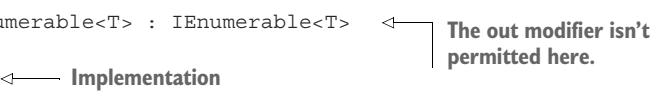
```
public TResult Func<in T, out TResult>(T arg)
```

In everyday development, you're likely to use existing variant interfaces and delegates more often than you declare them. A few restrictions exist in terms of the type arguments you can use. Let's look at them now.

4.4.3 Restrictions on using variance

To reiterate a point made earlier, variance can be declared only in interfaces and delegates. That variance isn't inherited by classes or structs implementing interfaces; classes and structs are always invariant. As an example, suppose you were to create a class like this:

```
public class SimpleEnumerable<T> : IEnumerable<T>
{
    // Implementation
}
```



The `out` modifier isn't permitted here.

You still couldn't convert from `SimpleEnumerable<string>` to a `SimpleEnumerable<object>`. You *could* convert from `SimpleEnumerable<string>` to `IEnumerable<object>` using the covariance of `IEnumerable<T>`.

Let's assume you're dealing with a delegate or interface with some covariant or contravariant type parameters. What conversions are available? You need definitions to explain the rules:

- A conversion involving variance is called a *variance conversion*.
- Variance conversion is one example of a *reference conversion*. A reference conversion is one that doesn't change the value involved (which is always a reference); it only changes the compile-time type.

- An *identity conversion* is a conversion from one type to the same type as far as the CLR is concerned. This might be the same type from a C# perspective, too (from `string` to `string`, for example), or it might be between types that are different only as far as the C# language is concerned, such as from `object` to `dynamic`.

Suppose you want to convert from `IEnumerable<A>` to `IEnumerable` for some type arguments A and B. That's valid if there's an identity or implicit reference conversion from A to B. For example, these conversions are valid:

- `IEnumerable<string>` to `IEnumerable<object>`: there's an implicit reference conversion from a class to its base class (or its base class's base class, and so forth).
- `IEnumerable<string>` to `IEnumerable<IConvertible>`: there's an implicit reference conversion from a class to any interface it implements.
- `IEnumerable<IDisposable>` to `IEnumerable<object>`: there's an implicit reference conversion from any reference type to `object` or `dynamic`.

These conversions are invalid:

- `IEnumerable<object>` to `IEnumerable<string>`: there's an *explicit* reference conversion from `object` to `string`, but not an *implicit* one.
- `IEnumerable<string>` to `IEnumerable<Stream>`: the `string` and `Stream` classes are unrelated.
- `IEnumerable<int>` to `IEnumerable<IConvertible>`: there's an implicit conversion from `int` to `IConvertible`, but it's a boxing conversion rather than a reference conversion.
- `IEnumerable<int>` to `IEnumerable<long>`: there's an implicit conversion from `int` to `long`, but it's a numeric conversion rather than a reference conversion.

As you can see, the requirement that the conversion between type arguments is a reference or identity conversion affects value types in a way that you might find surprising.

That example using `IEnumerable<T>` has only a single type argument to consider. What about when you have multiple type arguments? Effectively, they're checked pairwise from the source of the conversion to the target, making sure that each conversion is appropriate for the type parameter involved.

To put this more formally, consider a generic type declaration with n type parameters: `T<X1, ..., Xn>`. A conversion from `T<A1, ..., An>` to `T<B1, ..., Bn>` is considered in terms of each type parameter and pair of type arguments in turn. For each i between 1 and n:

- If X_i is covariant, there must be an identity or implicit reference conversion from A_i to B_i .
- If X_i is contravariant, there must be an identity or implicit reference conversion from B_i to A_i .
- If X_i is invariant, there must be an identity conversion from A_i to B_i .

To put this into a concrete example, let's consider `Func<in T, out TResult>`. The rules mean the following:

- There's a valid conversion from `Func<object, int>` to `Func<string, int>` because
 - The first type parameter is contravariant, and there's an implicit reference conversion from `string` to `object`.
 - The second type parameter is covariant, and there's an identity conversion from `int` to `int`.
- There's a valid conversion from `Func<dynamic, string>` to `Func<object, IConvertible>` because
 - The first type parameter is contravariant, and there's an identity conversion from `dynamic` to `object`.
 - The second type parameter is covariant, and there's an implicit reference conversion from `string` to `IConvertible`.
- There's no conversion from `Func<string, int>` to `Func<object, int>` because
 - The first type parameter is contravariant, and there's no implicit reference conversion from `object` to `string`.
 - The second type parameter doesn't matter; the conversion is already invalid because of the first type parameter.

Don't worry if all of this is a bit overwhelming; 99% of the time you won't even notice you're using generic variance. I've provided this detail to help you just in case you receive a compile-time error and don't understand why.³ Let's wrap up by looking at a couple of examples of when generic variance is useful.

4.4.4 Generic variance in practice

A lot of the time, you may end up using generic variance without even being conscious of doing so, because things just work as you'd probably want them to. There's no particular need to be aware that you're using generic variance, but I'll point out a couple of examples of where it's useful.

First, let's consider LINQ and `IEnumerable<T>`. Suppose you have strings that you want to perform a query on, but you want to end up with a `List<object>` instead of a `List<string>`. For example, you may need to add other items to the list afterward. The following listing shows how before covariance, the simplest way to do this would be to use an extra `Cast` call.

³ If this proves insufficient for a particular error, I suggest turning to the third edition, which has even more detail.

Listing 4.20 Creating a List<object> from a string query without variance

```
IEnumerable<string> strings = new[] { "a", "b", "cdefg", "hij" };
List<object> list = strings
    .Where(x => x.Length > 1)
    .Cast<object>()
    .ToList();
```

That feels annoying to me. Why create a whole extra step in the pipeline just to change the type in a way that'll always work? With variance, you can specify a type argument to the `ToList()` call instead, to specify the type of list you want, as in the following listing.

Listing 4.21 Creating a List<object> from a string query by using variance

```
IEnumerable<string> strings = new[] { "a", "b", "cdefg", "hij" };
List<object> list = strings
    .Where(x => x.Length > 1)
    .ToList<object>();
```

This works because the output of the `Where` call is an `IEnumerable<string>`, and you're asking the compiler to treat the input of the `ToList()` call as an `IEnumerable<object>`. That's fine because of variance.

I've found contravariance to be useful in conjunction with `IComparer<T>`, the interface for ordering comparisons of another type. As an example, suppose you have a `Shape` base class with an `Area` property and then `Circle` and `Rectangle` derived classes. You can write an `AreaComparer` that implements `IComparer<Shape>`, and that's fine for sorting a `List<Shape>` in place using `List<T>.Sort()`. But if you have a `List<Circle>` or a `List<Rectangle>`, how do you sort that? Various workarounds existed before generic variance, but the following listing shows how it's trivial now.

Listing 4.22 Sorting a List<Circle> with an IComparer<Shape>

```
List<Circle> circles = new List<Circle>
{
    new Circle(5.3),
    new Circle(2),
    new Circle(10.5)
};
circles.Sort(new AreaComparer());
foreach (Circle circle in circles)
{
    Console.WriteLine(circle.Radius);
}
```

The full source for the types used by listing 4.22 is in the downloadable code, but they're as simple as you'd expect them to be. The key point is that you can convert `AreaComparer` to `IComparer<Circle>` for the `Sort` method call. That wasn't the case before C# 4.

If you declare your own generic interfaces or delegates, it's always worth considering whether the type parameters can be covariant or contravariant. I wouldn't normally try to force the issue if it doesn't fall out that way naturally, but it's worth taking a moment to think about it. It can be annoying to use an interface that *could* have variant type parameters but where the developer just hadn't considered whether it might be useful to someone.

Summary

- C# 4 supports *dynamic typing*, which defers binding from compile time to execution time.
- Dynamic typing supports custom behavior via `IDynamicMetaObjectProvider` and the `DynamicObject` class.
- Dynamic typing is implemented with both compiler and framework features. The framework optimizes and caches heavily to make it reasonably efficient.
- C# 4 allows parameters to specify default values. Any parameter with a default value is an *optional parameter* and doesn't have to be provided by the caller.
- C# 4 allows arguments to specify the name of the parameter for which it's intended to provide the value. This works with optional parameters to allow you to specify arguments for some parameters but not others.
- C# 4 allows COM primary interop assemblies (PIAs) to be *linked* rather than *referenced*, which leads to a simpler deployment model.
- Linked PIAs expose variant values via dynamic typing, which avoids a lot of casting.
- Optional parameters are extended for COM libraries to allow `ref` parameters to be optional.
- Ref parameters in COM libraries can be specified by value.
- Generic variance allows safe conversions for generic interfaces and delegates based on whether values act as input or output.

Writing asynchronous code



This chapter covers

- What it means to write asynchronous code
- Declaring asynchronous methods with the `async` modifier
- Waiting asynchronously with the `await` operator
- Language changes in `async/await` since C# 5
- Following usage guidelines for asynchronous code

Asynchrony has been a thorn in the side of developers for years. It's been known to be useful as a way of avoiding tying up a thread while waiting for some arbitrary task to complete, but it's also been a pain in the neck to implement correctly.

Even within the .NET Framework (which is still relatively young in the grand scheme of things), we've had three models to try to make things simpler:

- The `BeginFoo/EndFoo` approach from .NET 1.x, using `IAsyncResult` and `AsyncCallback` to propagate results
- The event-based asynchronous pattern from .NET 2.0, as implemented by `BackgroundWorker` and `WebClient`

- The Task Parallel Library (TPL) introduced in .NET 4.0 and expanded in .NET 4.5

Despite the TPL's generally excellent design, writing robust and readable asynchronous code with it was hard. Although the support for parallelism was great, some aspects of general asynchrony are much better fixed in a language instead of purely in libraries.

The main feature of C# 5 is typically called *async/await*, and it builds on the TPL. It allows you to write synchronous-looking code that uses asynchrony where appropriate. Gone is the spaghetti of callbacks, event subscriptions, and fragmented error handling; instead, asynchronous code expresses its intentions clearly and in a form that builds on the structures that developers are already familiar with. The language construct introduced in C# 5 allows you to await an asynchronous operation. This awaiting looks very much like a normal blocking call in that the rest of your code won't continue until the operation has completed, but it manages to do this without blocking the currently executing thread. Don't worry if that statement sounds completely contradictory; all will become clear over the course of the chapter.

Async/await has evolved a little over time, and for simplicity I've included the new features from C# 6 and C# 7 alongside the original C# 5 descriptions. I've called out those changes so you know when you need a C# 6 or C# 7 compiler.

The .NET Framework embraced asynchrony wholeheartedly in version 4.5, exposing asynchronous versions of a great many operations following a *task-based asynchronous pattern* to give a consistent experience across multiple APIs. Similarly, the Windows Runtime platform, which is the basis of Universal Windows Applications (UWA/UWP), enforces asynchrony for all long-running (or potentially long-running) operations. Many other modern APIs rely heavily on asynchrony, such as Roslyn and `HttpClient`. In short, most C# developers will have to use asynchrony in at least *some* part of their work.

NOTE The Windows Runtime platform is commonly known as *WinRT*; it's not to be confused with Windows RT, which was an edition of Windows 8.x for ARM processors. *Universal Windows Applications* are an evolution of Windows Store applications. *UWP* is a further evolution of UWA from Windows 10 onward.

To be clear, C# hasn't become omniscient, guessing where you might want to perform operations concurrently or asynchronously. The compiler is smart, but it doesn't attempt to remove the *inherent* complexity of asynchronous execution. You still need to think carefully, but the beauty of *async/await* is that all the tedious and confusing boilerplate code that used to be required has gone. Without the distraction of all the fluff required to make your code asynchronous to start with, you can concentrate on the hard bits.

A word of warning: this topic is reasonably advanced. It has the unfortunate properties of being incredibly important (realistically, even entry-level developers need to have a reasonable understanding of it) but also quite tricky to get your head around to start with.

This chapter focuses on asynchrony from a “regular developer” perspective, so you can use `async/await` without needing to understand too much of the detail. Chapter 6 goes into a lot more of the complexity of the implementation. I feel you’ll be a better developer if you understand what’s going on behind the scenes, but you can certainly take what you’ll learn from this chapter and be productive with `async/await` before diving deeper. Even within this chapter, you’ll be looking at the feature in an iterative process, with more detail the further you go.

5.1 Introducing asynchronous functions

So far, I’ve claimed that C# 5 makes `async` easier, but I’ve given only a tiny description of the features involved. Let’s fix that and then look at an example.

C# 5 introduces the concept of an *asynchronous function*. This is always either a method or an anonymous function that’s declared with the `async` modifier, and it can use the `await` operator for await expressions.

NOTE As a reminder, an anonymous function is either a lambda expression or an anonymous method.

The *await expressions* are the points where things get interesting from a language perspective: if the operation the expression is awaiting hasn’t completed yet, the asynchronous function will return immediately, and it’ll then continue where it left off (in an appropriate thread) when the value becomes available. The natural flow of not executing the next statement until this one has completed is still maintained but without blocking. I’ll break down that woolly description into more-concrete terms and behavior later, but you need to see an example before it’s likely to make any sense.

5.1.1 First encounters of the asynchronous kind

Let’s start with something simple that demonstrates asynchrony in a practical way. We often curse network latency for causing delays in our real applications, but latency does make it easy to show why asynchrony is so important—particularly when using a GUI framework such as Windows Forms. Our first example is a tiny Windows Forms app that fetches the text of this book’s homepage and displays the length of the HTML in a label.

Listing 5.1 Displaying a page length asynchronously

```
public class AsyncIntro : Form
{
    private static readonly HttpClient client = new HttpClient();
    private readonly Label label;
    private readonly Button button;
```

```

public AsyncIntro()
{
    label = new Label
    {
        Location = new Point(10, 20),
        Text = "Length"
    };
    button = new Button
    {
        Location = new Point(10, 50),
        Text = "Click"
    };
    button.Click += DisplayWebSiteLength;
    AutoSize = true;
    Controls.Add(label);
    Controls.Add(button);
}

async void DisplayWebSiteLength(object sender, EventArgs e)
{
    label.Text = "Fetching...";
    string text = await client.GetStringAsync(
        "http://csharpindepth.com");
    label.Text = text.Length.ToString();    ← | Updates
}                                         | the UI

static void Main()
{
    Application.Run(new AsyncIntro());    ← | Entry point; just
}                                         | runs the form
}

```

The first part of this code creates the UI and hooks up an event handler for the button in a straightforward way. It's the `DisplayWebSiteLength` method that's of interest here. When you click the button, the text of the homepage is fetched, and the label is updated to display the HTML length in characters.

NOTE I'm not disposing of the task returned by `GetStringAsync`, even though `Task` implements `IDisposable`. Fortunately, you don't need to dispose of tasks in general. The background of this is somewhat complicated, but Stephen Toub explains it in a blog post dedicated to the topic: <http://mng.bz/E6L3>.

I could've written a smaller example program as a console app, but hopefully listing 5.1 makes a more convincing demo. In particular, if you remove the `async` and `await` contextual keywords, change `HttpClient` to `WebClient`, and change `GetStringAsync` to `DownloadString`, the code will still compile and work, but the UI will freeze while it fetches the contents of the page. If you run the `async` version (ideally, over a slow network connection), you'll see that the UI is responsive; you can still move the window around while the web page is fetching.

NOTE `HttpClient` is in some senses the new and improved `WebClient`; it's the preferred HTTP API for .NET 4.5 onward, and it contains only asynchronous operations.

Most developers are familiar with the two golden rules of threading in Windows Forms development:

- Don't perform any time-consuming action on the UI thread.
- Don't access any UI controls *other* than on the UI thread.

You may regard Windows Forms as a legacy technology these days, but most GUI frameworks have the same rules, and they're easier to state than to obey. As an exercise, you might want to try a few ways of creating code similar to listing 5.1 without using `async/await`. For this extremely simple example, it's not too bad to use the event-based `WebClient.DownloadStringAsync` method, but as soon as more complex flow control (error handling, waiting for multiple pages to complete, and so on) comes into the equation, the legacy code quickly becomes hard to maintain, whereas the C# 5 code can be modified in a natural way.

At this point, the `DisplayWebSiteLength` method feels somewhat magical: you know it does what you need it to, but you have no idea how. Let's take it apart a little bit and save the gory details for later.

5.1.2 **Breaking down the first example**

You'll start by slightly expanding the method. In listing 5.1, I used `await` directly on the return value of `HttpClient.GetStringAsync`, but you can separate the call from the awaiting part:

```
async void DisplayWebSiteLength(object sender, EventArgs e)
{
    label.Text = "Fetching...";
    Task<string> task = client.GetStringAsync("http://csharpindepth.com");
    string text = await task;
    label.Text = text.Length.ToString();
}
```

Notice that the type of `task` is `Task<string>`, but the type of the `await task` expression is simply `string`. In this sense, the `await` operator performs an unwrapping operation—at least when the value being awaited is a `Task<TResult>`. (As you'll see, you can await other types, too, but `Task<TResult>` is a good starting point.) That's one aspect of `await` that doesn't seem directly related to asynchrony but makes life easier.

The main purpose of `await` is to avoid blocking while you wait for time-consuming operations to complete. You may be wondering how this all works in the concrete terms of threading. You're setting `label.Text` at the start and end of the method, so it's reasonable to assume that both of those statements are executed on the UI thread, and yet you're clearly not blocking the UI thread while you wait for the web page to download.

The trick is that the method returns as soon as it hits the `await` expression. Until that point, it executes synchronously on the UI thread, like any other event handler

would. If you put a breakpoint on the first line and hit it in the debugger, you'll see that the stack trace shows that the button is busy raising its `Click` event, including the `Button.OnClick` method. When you reach the `await`, the code checks whether the result is already available, and if it's not (which will almost certainly be the case), it schedules a continuation to be executed when the web operation has completed. In this example, the continuation executes the rest of the method, effectively jumping to the end of the `await` expression. The continuation is executed in the UI thread, which is what you need so that you can manipulate the UI.

DEFINITION A *continuation* is effectively a callback to be executed when an asynchronous operation (or any `Task`) has completed. In an `async` method, the continuation maintains the state of the method. Just as a closure maintains its environment in terms of variables, a continuation remembers the point where it reached, so it can continue from there when it's executed. The `Task` class has a method specifically for attaching continuations: `Task.ContinueWith`.

If you then put a breakpoint in the code after the `await` expression and run the code again, then assuming that the `await` expression needed to schedule the continuation, you'll see that the stack trace no longer has the `Button.OnClick` method in it. That method finished executing long ago. The call stack will now effectively be the bare Windows Forms event loop with a few layers of `async` infrastructure on top. The call stack will be similar to what you'd see if you called `Control.Invoke` from a background thread in order to update the UI appropriately, but it's all been done for you. At first it can be unnerving to notice the call stack change dramatically under your feet, but it's absolutely necessary for asynchrony to be effective.

The compiler achieves all of this by creating a complicated state machine. That's an implementation detail you'll look at in chapter 6, but for now you're going to concentrate on the functionality that `async/await` provides. First, you need a more concrete description of what you're trying to achieve and what the language specifies.

5.2 Thinking about asynchrony

If you ask a developer to describe asynchronous execution, chances are they'll start talking about multithreading. Although that's an important part of *typical* uses of asynchrony, it's not required for asynchronous execution. To fully appreciate how the `async` feature of C# 5 works, it's best to strip away any thoughts of threading and go back to basics.

5.2.1 Fundamentals of asynchronous execution

Asynchrony strikes at the very heart of the execution model that C# developers are familiar with. Consider simple code like this:

```
Console.WriteLine("First");
Console.WriteLine("Second");
```

You expect the first call to complete and then the second call to start. Execution flows from one statement to the next, in order. But an asynchronous execution model doesn't work that way. Instead, it's all about *continuations*. When you start doing something, you tell that operation what you want to happen when that operation has completed. You may have heard (or used) the term *callback* for the same idea, but that has a broader meaning than the one we're after here. In the context of asynchrony, I'm using the term to refer to callbacks that preserve the state of the program rather than arbitrary callbacks for other purposes, such as GUI event handlers.

Continuations are naturally represented as delegates in .NET, and they're typically actions that receive the results of the asynchronous operation. That's why, to use the asynchronous methods in `WebClient` prior to C# 5, you'd wire up various events to say what code should be executed in the case of success, failure, and so on. The trouble is, creating all those delegates for a complicated sequence of steps ends up being very complicated, even with the benefit of lambda expressions. It's even worse when you try to make sure that your error handling is correct. (On a good day, I can be reasonably confident that the success paths of handwritten asynchronous code are correct. I'm typically less certain that it reacts the right way on failure.)

Essentially, all that `await` in C# does is ask the compiler to build a continuation for you. For an idea that can be expressed so simply, however, the consequences for readability and developer serenity are remarkable.

My earlier description of asynchrony was an idealized one. The reality in the task-based asynchronous pattern is slightly different. Instead of the continuation being passed to the asynchronous operation, the asynchronous operation starts and returns a token you can use to provide the continuation later. It represents the ongoing operation, which may have completed before it has returned to the calling code or may still be in progress. That token is then used whenever you want to express this idea: I can't proceed any further until this operation has completed. Typically, the token is in the form of a `Task` or `Task<TResult>`, but it doesn't have to be.

NOTE The token described here isn't the same as a cancellation token, although both have the same emphasis on the fact that you don't need to know what's going on behind the scenes; you only need to know what the token allows you to do.

The execution flow in an asynchronous method in C# 5 typically follows these lines:

- 1 Do some work.
- 2 Start an asynchronous operation and remember the token it returns.
- 3 Possibly do some more work. (Often, you can't make any further progress until the asynchronous operation has completed, in which case this step is empty.)
- 4 Wait for the asynchronous operation to complete (via the token).
- 5 Do some more work.
- 6 Finish.

If you didn't care about exactly what the wait part meant, you could do all of this in C# 4. If you're happy to *block* until the asynchronous operation completes, the token will normally provide you some way of doing so. For a Task, you could simply call `Wait()`. At that point, though, you're taking up a valuable resource (a thread) and not doing any useful work. It's a little like phoning for a delivery pizza and then standing at your front door until it arrives. What you really want to do is get on with something else and ignore the pizza until it arrives. That's where `await` comes in.

When you wait for an asynchronous operation, you're saying "I've gone as far as I can go for now. Keep going when the operation has completed." But if you're not going to block the thread, what can you do? Very simply, you can return right then and there. You'll continue asynchronously yourself. And if you want your caller to know when your asynchronous method has completed, you'll pass a token back to the caller, which they can block on if they want or (more likely) use with another continuation. Often, you'll end up with a whole stack of asynchronous methods calling each other; it's almost as if you go into an "async mode" for a section of code. Nothing in the language states that it has to be done that way, but the fact that the same code that consumes asynchronous operations also behaves as an asynchronous operation certainly encourages it.

5.2.2 Synchronization contexts

Earlier I mentioned that one of the golden rules of UI code is that you mustn't update the user interface unless you're on the right thread. In listing 5.1, which checked the length of a web page asynchronously, you needed to ensure that the code after the `await` expression executed on the UI thread. Asynchronous functions get back to the right thread by using `SynchronizationContext`, a class that's existed since .NET 2.0 and is used by other components such as `BackgroundWorker`. A `SynchronizationContext` generalizes the idea of executing a delegate on an appropriate thread; its `Post` (asynchronous) and `Send` (synchronous) messages are similar to `Control.BeginInvoke` and `Control.Invoke` in Windows Forms.

Different execution environments use different contexts; for example, one context may let any thread from the thread pool execute the action it's given. More contextual information exists than in the synchronization context, but if you start wondering how asynchronous methods manage to execute exactly where you want them to, it's the synchronization context that you need to focus on.

For more information on `SynchronizationContext`, read Stephen Cleary's MSDN magazine article on the topic (<http://mng.bz/5cDw>). In particular, pay careful attention if you're an ASP.NET developer; the ASP.NET context can easily trap unwary developers into creating deadlocks within code that looks fine. The story changes slightly for ASP.NET Core, but Stephen has another blog post covering that: <http://mng.bz/5YrO>.

Use of Task.Wait() and Task.Result in examples

I've used `Task.Wait()` and `Task.Result` in some of the sample code because it leads to simple examples. It's usually safe to do so in a console application because in that case there's no synchronization context; continuations for async methods will always execute in the thread pool.

In real-world applications, you should take great care using these methods. They both block until they complete, which means if you call them from a thread that a continuation needs to execute on, you can easily deadlock your application.

With the theory out of the way, let's take a closer look at the concrete details of asynchronous methods. Asynchronous anonymous functions fit into the same mental model, but it's much easier to talk about asynchronous methods.

5.2.3 Modeling asynchronous methods

I find it useful to think about asynchronous methods as shown in figure 5.1.

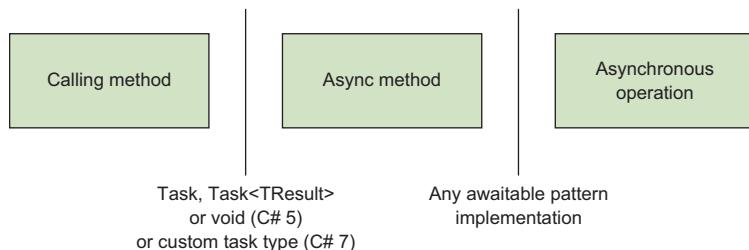


Figure 5.1 Modeling asynchronous boundaries

Here you have three blocks of code (the methods) and two boundary types (the method return types). As a simple example, in a console-based version of our page-length fetching application, you might have code like the following.

Listing 5.2 Retrieving a page length in an asynchronous method

```

static readonly HttpClient client = new HttpClient();

static async Task<int> GetPageLengthAsync(string url)
{
    Task<string> fetchTextTask = client.GetStringAsync(url);
    int length = (await fetchTextTask).Length;
    return length;
}

```

```
static void PrintPageLength()
{
    Task<int> lengthTask =
        GetPageLengthAsync("http://csharpindepth.com");
    Console.WriteLine(lengthTask.Result);
}
```

Figure 5.2 shows how the concrete details in listing 5.2 map to the concepts in figure 5.1.

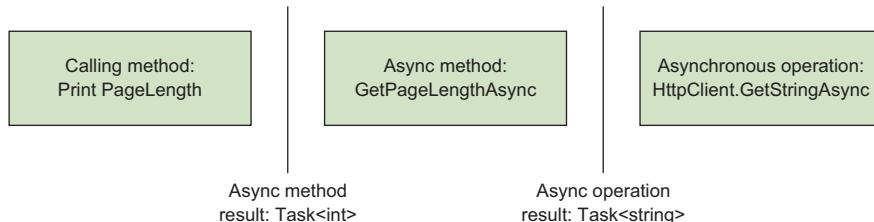


Figure 5.2 Applying the details of listing 5.2 to the general pattern shown in figure 5.1

You're mainly interested in the `GetPageLengthAsync` method, but I've included `PrintPageLength` so you can see how the methods interact. In particular, you definitely need to know about the valid types at the method boundaries. I'll repeat this diagram in various forms through the chapter.

You're finally ready to look at writing async methods and the way they'll behave. There's a lot to cover here, as what you can do and what happens when you do it blend together to a large extent.

There are only two new pieces of syntax: `async` is a modifier used when declaring an asynchronous method, and the `await` operator is used to consume asynchronous operations. But following the way information is transferred between parts of your program gets complicated quickly, especially when you have to consider what happens when things go wrong. I've tried to separate out the different aspects, but your code will be dealing with everything at once. If you find yourself asking “But what about...?” while reading this section, keep reading; chances are your question will be answered soon.

The next three sections look at an asynchronous method in three stages:

- Declaring the `async` method
- Using the `await` operator to asynchronously wait for operations to complete
- Returning a value when your method is complete

Figure 5.3 shows how these sections fit into our conceptual model.

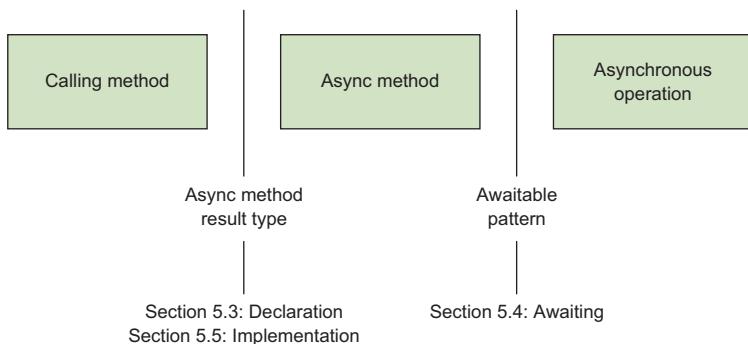


Figure 5.3 Demonstrating how sections 5.3, 5.4, and 5.5 fit into the conceptual model of asynchrony

Let's start with the method declaration itself; that's the easiest bit.

5.3 **Async method declarations**

The syntax for an `async` method declaration is exactly the same as for any other method, except it has to include the `async` contextual keyword. This can appear anywhere before the return type. All of these are valid:

```

public static async Task<int> FooAsync() { ... }
public async static Task<int> FooAsync() { ... }
async public Task<int> FooAsync() { ... }
public async virtual Task<int> FooAsync() { ... }
  
```

My preference is to keep the `async` modifier immediately before the return type, but there's no reason you shouldn't come up with your own convention. As always, discuss it with your team and try to be consistent within one codebase.

Now, the `async` contextual keyword has a little secret: the language designers didn't need to include it at all. In the same way the compiler goes into a sort of iterator block mode when you try to use `yield return` or `yield break` in a method with a suitable return type, the compiler could have spotted the use of `await` inside a method and used that to go into `async` mode. But I'm pleased that `async` is required, because it makes it much easier to read code written using asynchronous methods. It sets your expectations immediately, so you're actively looking for `await` expressions, and you can actively look for any blocking calls that should be turned into an `async` call and an `await` expression.

The fact that the `async` modifier has no representation in the generated code is important, though. As far as the calling method is concerned, it's a normal method that happens to return a task. You can change an existing method (with an appropriate

signature) to use `async`, or you could go in the other direction; it's a compatible change in terms of both source and binary. The fact that it's a detail of the implementation of the method means that you can't declare an abstract method or a method in an interface using `async`. It's perfectly possible for there to be an interface specifying a method with a return type of `Task<int>`; one implementation of that interface can use `async/await` while another implementation uses a regular method.

5.3.1 Return types from `async` methods

Communication between the caller and the `async` method is effectively in terms of the value returned. In C# 5, asynchronous functions are limited to the following return types:

- `void`
- `Task`
- `Task<TResult>` (for some type `TResult`, which could itself be a type parameter)

In C# 7, this list is expanded to include *task types*. You'll come back to those in section 5.8 and then again in chapter 6.

The .NET 4 `Task` and `Task<TResult>` types both represent an operation that may not have completed yet; `Task<TResult>` derives from `Task`. The difference between the two is that `Task<TResult>` represents an operation that returns a value of type `TResult`, whereas `Task` need not produce a result at all. It's still useful to return a `Task`, though, because it allows the calling code to attach its own continuations to the returned task, detect when the task has failed or completed, and so on. In some cases, you can think of `Task` as being like a `Task<void>` type, if such a thing were valid.

NOTE F# developers can be justifiably smug about the `Unit` type at this point, which is similar to `void` but is a real type. The disparity between `Task` and `Task<TResult>` can be frustrating. If you could use `void` as a type argument, you wouldn't need the `Action` family of delegates either; `Action<string>` is equivalent to `Func<string, void>`, for example.

The ability to return `void` from an `async` method is designed for compatibility with event handlers. For example, you might have a UI button click handler like this:

```
private async void LoadStockPrice(object sender, EventArgs e)
{
    string ticker = tickerInput.Text;
    decimal price = await stockPriceService.FetchPriceAsync(ticker);
    priceDisplay.Text = price.ToString("c");
}
```

This is an asynchronous method, but the calling code (the button `OnClick` method or whatever piece of framework code is raising the event) doesn't care. It doesn't need to know when you've finished handling the event—when you've loaded the stock price and updated the UI. It simply calls the event handler that it's been given. The fact that

the code generated by the compiler will end up with a state machine attaching a continuation to whatever is returned by `FetchPriceAsync` is an implementation detail.

You can subscribe to an event with the preceding method as if it were any other event handler:

```
loadStockPriceButton.Click += LoadStockPrice;
```

After all (and yes, I'm laboring this deliberately), it's just a normal method as far as calling code is concerned. It has a `void` return type and parameters of type `object` and `EventArgs`, which makes it suitable as the action for an `EventHandler` delegate instance.

WARNING Event subscription is pretty much the only time I'd recommend returning `void` from an asynchronous method. Any other time you don't need to return a specific value, it's best to declare the method to return `Task`. That way, the caller is able to await the operation completing, detect failures, and so on.

Although the return type of `async` methods is fairly tightly restricted, most other aspects are as normal: `async` methods can be generic, static or nonstatic, and specify any of the regular access modifiers. Restrictions exist on the parameters you can use, however.

5.3.2 **Parameters in `async` methods**

None of the parameters in an `async` method can use the `out` or `ref` modifiers. This makes sense because those modifiers are for communicating information back to the calling code; some of the `async` method may not have run by the time control returns to the caller, so the value of the by-reference parameter might not have been set. Indeed, it could get stranger than that: imagine passing a local variable as an argument for a `ref` parameter; the `async` method could end up trying to set that variable after the calling method had already completed. It doesn't make a lot of sense to try to do this, so the compiler prohibits it. Additionally, pointer types can't be used as `async` method parameter types.

After you've declared the method, you can start writing the body and awaiting other asynchronous operations. Let's look at how and where you can use `await` expressions.

5.4 **Await expressions**

The whole point of declaring a method with the `async` modifier is to use `await` expressions in that method. Everything else about the method looks pretty normal: you can use all kinds of control flow—loops, exceptions, `using` statements, anything. So where can you use an `await` expression, and what does it do?

The syntax for an `await` expression is simple: it's the `await` operator followed by another expression that produces a value. You can await the result of a method call, a variable, a property. It doesn't have to be a simple expression either. You can chain method calls together and await the result:

```
int result = await foo.Bar().Baz();
```

The precedence of the `await` operator is lower than that of the dot operator, so this code is equivalent to the following:

```
int result = await (foo.Bar().Baz());
```

Restrictions limit which expressions you can await, though. They have to be *awaitable*, and that's where the awaitable pattern comes in.

5.4.1 The awaitable pattern

The *awaitable pattern* is used to determine types that can be used with the `await` operator. Figure 5.4 is a reminder that I'm talking about the second boundary from figure 5.1: how the `async` method interacts with another asynchronous operation. The awaitable pattern is a way of codifying what we mean by an *asynchronous operation*.

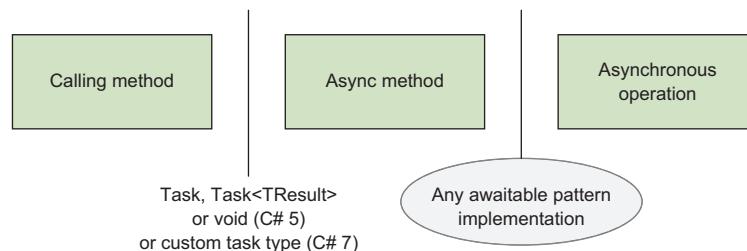


Figure 5.4 The awaitable pattern enables `async` methods to asynchronously wait for operations to complete

You might expect this to be expressed in terms of interfaces in the same way the compiler requires a type to implement `IDisposable` in order to support the `using` statement. Instead, it's based on a pattern. Imagine that you have an expression of type `T` that you want to await. The compiler performs the following checks:

- `T` must have a parameterless `GetAwaiter()` instance method, or there must be an extension method accepting a single parameter of type `T`. The `GetAwaiter` method has to be nonvoid. The return type of the method is called the *awaiter type*.
- The awainer type must implement the `System.Runtime.INotifyCompletion` interface. That interface has a single method: `void OnCompleted (Action)`.
- The awainer type must have a readable instance property called `IsCompleted` of type `bool`.
- The awainer type must have a nongeneric parameterless instance method called `GetResult`.
- The members listed previously don't have to be public, but they need to be accessible from the `async` method you're trying to await the value from. (Therefore, it's possible that you can await a value of a particular type from some code but not in all code. That's highly unusual, though.)

If `T` passes all of those checks, congratulations—you can await a value of type `T!` The compiler needs one more piece of information, though, to determine what the type of the await expression should be. That's determined by the return type of the `GetResult` method of theawaiter type. It's fine for it to be a `void` method, in which case the await expression is classified as an expression with no result, like an expression that calls a `void` method directly. Otherwise, the await expression is classified as producing a value of the same type as the return type of `GetResult`.

As an example, let's consider the static `Task.Yield()` method. Unlike most other methods on `Task`, the `Yield()` method doesn't return a task itself; it returns a `YieldAwaitable`. Here's a simplified version of the types involved:

```
public class Task
{
    public static YieldAwaitable Yield();
}

public struct YieldAwaitable
{
    public YieldAwaiter GetAwaiter();

    public struct YieldAwaiter : INotifyCompletion
    {
        public bool IsCompleted { get; }
        public void OnCompleted(Action continuation);
        public void GetResult();
    }
}
```

As you can see, `YieldAwaitable` follows the awaitable pattern described previously. Therefore, this is valid:

```
public async Task ValidPrintYieldPrint()
{
    Console.WriteLine("Before yielding");
    await Task.Yield();           ←———— Valid
    Console.WriteLine("After yielding");
}
```

But the following is invalid, because it tries to use the result of awaiting a `YieldAwaitable`:

```
public async Task InvalidPrintYieldPrint()
{
    Console.WriteLine("Before yielding");
    var result = await Task.Yield();   ←———— Invalid; this await expression
    Console.WriteLine("After yielding"); doesn't produce a value.
}
```

The middle line of `InvalidPrintYieldPrint` is invalid for exactly the same reason that it would be invalid to write this:

```
var result = Console.WriteLine("WriteLine is a void method");
```

No result is produced, so you can't assign it to a variable.

Unsurprisingly, the awainer type for Task has a `GetResult` method with a `void` return type, whereas the awainer type for `Task<TResult>` has a `GetResult` method returning `TResult`.

Historical importance of extension methods

The fact that `GetAwaiter` can be an extension method is of more historical than contemporary importance. C# 5 was released in the same time frame as .NET 4.5, which introduced the `GetAwaiter` methods into `Task` and `Task<TResult>`. If `GetAwaiter` had to be a genuine instance method, that would've stranded developers who were tied to .NET 4.0. But with support for extension methods, `Task` and `Task<TResult>` could be `async/await`-enabled by using a NuGet package to provide those extension methods separately. This also meant that the community could test prereleases of the C# 5 compiler without testing prereleases of .NET 4.5.

In code targeting modern frameworks in which all the relevant `GetAwaiter` methods are already present, you'll rarely need to use the ability to make an existing type awaitable via extension methods.

You'll see more details about exactly how the members in the awaitable pattern are used in section 5.6, when you consider the execution flow of asynchronous methods. You're not quite done with `await` expressions, though; a few restrictions exist.

5.4.2 Restrictions on await expressions

Like `yield return`, restrictions limit where you can use `await` expressions. The most obvious restriction is that you can use them only in `async` methods and `async` anonymous functions (which you'll look at in section 5.7). Even within `async` methods, you can't use the `await` operator within an anonymous function unless that's `async`, too.

The `await` operator also isn't allowed within an unsafe context. That doesn't mean you can't use unsafe code within an `async` method; you just can't use the `await` operator within that part. The following listing shows a contrived example in which a pointer is used to iterate over the characters in a string to find the total of the UTF-16 code units in that string. It doesn't do anything truly useful, but it demonstrates the use of an unsafe context within an `async` method.

Listing 5.3 Using unsafe code in an `async` method

```
static async Task DelayWithResultOfUnsafeCode(string text)
{
    int total = 0;
    unsafe
    {
        fixed (char* textPointer = text)
        {
            char* p = textPointer;
            // ...
        }
    }
}
```

It's fine to have an unsafe context in an `async` method.

```

        while (*p != 0)
        {
            total += *p;
            p++;
        }
    }
Console.WriteLine("Delaying for " + total + "ms");
await Task.Delay(total);
Console.WriteLine("Delay complete");
}

```

← But, the await expression
can't be inside it.

You also can't use the `await` operator within a lock. If you ever find yourself wanting to hold a lock while an asynchronous operation completes, you should redesign your code. Don't work around the compiler restriction by calling `Monitor.TryEnter` and `Monitor.Exit` manually with a `try/finally` block; change your code so you don't need the lock during the operation. If this is really, really awkward in your situation, consider using `SemaphoreSlim` instead, with its `WaitAsync` method.

The monitor used by a `lock` statement can be released only by the same thread that originally acquired it, which goes against the distinct possibility that the thread executing the code before an `await` expression will be different from the one executing the code after it. Even if the same thread is used (for example, because you're in a GUI synchronization context), some other code may well have executed on the same thread between the start and end of the asynchronous operation, and that other code would've been able to enter a `lock` statement for the same monitor, which almost certainly isn't what you intended. Basically, `lock` statements and asynchrony don't go well together.

There's one final set of contexts in which the `await` operator was invalid in C# 5 but is valid from C# 6 onward:

- Any `try` block with a `catch` block
- Any `catch` block
- Any `finally` block

It's always been okay to use the `await` operator in a `try` block that has only a `finally` block, which means it's always been okay to use `await` in a `using` statement. The C# design team didn't figure out how to safely and reliably include `await` expressions in the contexts listed previously before C# 5 shipped. This was occasionally inconvenient, and the team worked out how to build the appropriate state machine while implementing C# 6, so the restriction is lifted there.

You now know how to declare an `async` method and how the `await` operator can be used within it. What about when you've completed your work? Let's look at how values are returned back to the calling code.

5.5 Wrapping of return values

We've looked at how to declare the boundary between the calling code and the `async` method and how to wait for any asynchronous operations within the `async` method.

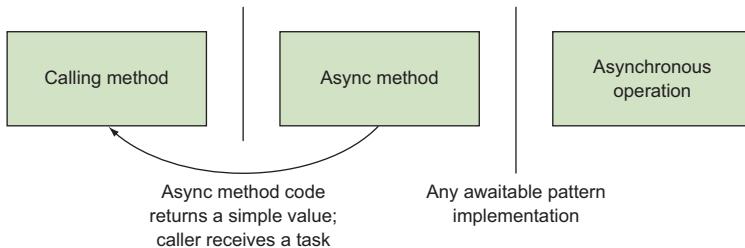


Figure 5.5 Returning a result from an async method to its caller

Now let's look at how return statements are used to implement that first boundary in terms of returning a value to the calling code; see figure 5.5.

You've already seen an example that returned data, but let's look at it again, this time focusing on the return aspect alone. Here's the relevant part of listing 5.2:

```
static async Task<int> GetPageLengthAsync(string url)
{
    Task<string> fetchTextTask = client.GetStringAsync(url);
    int length = (await fetchTextTask).Length;
    return length;
}
```

You can see that the type of `length` is `int`, but the return type of the method is `Task<int>`. The generated code takes care of the wrapping for you, so the caller gets a `Task<int>`, which will eventually have the value returned from the method when it completes. A method returning a nongeneric `Task` is like a normal `void` method: it doesn't need a return statement at all, and any return statements it does have must be simply `return` rather than trying to specify a value. In either case, the task will also propagate any exception thrown within the `async` method. (You'll look at exceptions in more detail in section 5.6.5.)

Hopefully, by now you should have a good intuition about why this wrapping is necessary: the method will almost certainly return to the caller before it hits the `return` statement, and it has to propagate the information to that caller somehow. A `Task<TResult>` (often known as a *future* in computer science) is the promise of a value—or an exception—at a later time.

As with normal execution flow, if the `return` statement occurs within the scope of a `try` block that has an associated `finally` block (including when all of this happens because of a `using` statement), the expression used to compute the return value is evaluated immediately, but it doesn't become the result of the task until everything has been cleaned up. If the `finally` block throws an exception, you don't get a task that both succeeds and fails; the whole thing will fail.

To reiterate a point I made earlier, it's the combination of automatic wrapping and unwrapping that makes the `async` feature work so well with composition; `async` methods

can consume the results of `async` methods easily, so you can build up complex systems from lots of small blocks. You can think of this as being a bit like LINQ; you write operations on each element of a sequence in LINQ, and the wrapping and unwrapping means you can apply those operations to sequences and get sequences back. In an `async` world, you rarely need to explicitly handle a task; instead, you `await` the task to consume it, and produce a result task automatically as part of the mechanism of the `async` method. Now that you know what an asynchronous method looks like, it's easier to give examples to demonstrate the execution flow.

5.6 Asynchronous method flow

You can think about `async/await` at multiple levels:

- You can simply expect that `awaiting` will do what you want without defining exactly what that means.
- You can reason about how the code will execute, in terms of what happens when and in which thread, but without understanding how that's achieved.
- You can dig deeply into the infrastructure that makes all of this happen.

So far, we've mostly been thinking at the first level, dipping down to the second occasionally. This section focuses on the second level, effectively looking at what the language promises. We'll leave the third bullet to the next chapter, where you'll see what the compiler is doing under the covers. (Even then, you could always go further; this book doesn't talk about anything below the IL level. We don't get into the operating system or hardware support for asynchrony and threading.)

For the vast majority of the time when you're developing, it's fine to switch between the first two levels, depending on your context. Unless I'm writing code that's coordinating multiple operations, I rarely need to even think at the second level of detail. Most of the time, I'm happy to just let things work. What's important is that you can think about the details when you need to.

5.6.1 What is awaited and when?

Let's start by simplifying things a bit. Sometimes `await` is used with the result of a chained method call or occasionally a property, like this:

```
string pageText = await new HttpClient().GetStringAsync(url);
```

This makes it look as if `await` can modify the meaning of the whole expression. In reality, `await` always operates on only a single value. The preceding line is equivalent to this:

```
Task<string> task = new HttpClient().GetStringAsync(url);
string pageText = await task;
```

Similarly, the result of an `await` expression can be used as a method argument or within another expression. Again, it helps if you can mentally separate out the `await`-specific part from everything else.

Imagine you have two methods, `GetHourlyRateAsync()` and `GetHoursWorkedAsync()`, returning a `Task<decimal>` and a `Task<int>`, respectively. You might have this complicated statement:

```
AddPayment(await employee.GetHourlyRateAsync() *  
           await timeSheet.GetHoursWorkedAsync(employee.Id));
```

The normal rules of C# expression evaluation apply, and the left operand of the `*` operator has to be completely evaluated before the right operand is evaluated, so the preceding statement can be expanded as follows:

```
Task<decimal> hourlyRateTask = employee.GetHourlyRateAsync();  
decimal hourlyRate = await hourlyRateTask;  
Task<int> hoursWorkedTask = timeSheet.GetHoursWorkedAsync(employee.Id);  
int hoursWorked = await hoursWorkedTask;  
AddPayment(hourlyRate * hoursWorked);
```

How you write the code is a different matter. If you find the single statement version easier to read, that's fine; if you want to expand it all out, you'll end up with more code, but it may be simpler to understand and debug. You could decide to use a third form that looks similar but isn't quite the same:

```
Task<decimal> hourlyRateTask = employee.GetHourlyRateAsync();  
Task<int> hoursWorkedTask = timeSheet.GetHoursWorkedAsync(employee.Id);  
AddPayment(await hourlyRateTask * await hoursWorkedTask);
```

I find that this is the most readable form, and it has potential performance benefits, too. You'll come back to this example in section 5.10.2.

The key takeaway from this section is that you need to be able to work out what's being awaited and when. In this case, the tasks returned from `GetHourlyRateAsync` and `GetHoursWorkedAsync` are being awaited. In every case, they're being awaited before the call to `AddPayment` is executed, which makes sense, because you need the intermediate results so you can multiply them together and pass the result of that multiplication as an argument. If this were using synchronous calls, all of this would be obvious; my aim is to demystify the awaiting part. Now that you know how to simplify complex code into the value you're awaiting and when you're awaiting it, you can move on to what happens when you're in the awaiting part itself.

5.6.2 Evaluation of await expressions

When execution reaches the `await` expression, you have two possibilities: either the asynchronous operation you're awaiting has already completed or it hasn't. If the operation has already completed, the execution flow is simple: it keeps going. If the operation failed and it captured an exception to represent that failure, the exception is thrown. Otherwise, any result from the operation is obtained (for example, extracting the `string` from a `Task<string>`) and you move on to the next part of the program. All of this is done without any thread context switching or attaching continuations to anything.

In the more interesting scenario, the asynchronous operation is still ongoing. In this case, the method waits asynchronously for the operation to complete and then continues in an appropriate context. This asynchronous waiting really means the method isn't executing at all. A continuation is attached to the asynchronous operation, and the method returns. The async infrastructure makes sure that the continuation executes on the right thread: typically, either a thread-pool thread (where it doesn't matter which thread is used) or the UI thread where that makes sense. This depends on the synchronization context (discussed in section 5.2.2) and can also be controlled using `Task.ConfigureAwait`, which we'll talk about in section 5.10.1.

Returning vs. completing

Possibly the hardest part of describing asynchronous behavior is talking about when the method *returns* (either to the original caller or to whatever called a continuation) and when the method *completes*. Unlike most methods, an asynchronous method can return multiple times—effectively, when it has no more work it can do for the moment.

To return to our earlier pizza delivery analogy, if you have an `EatPizzaAsync` method that involves calling the pizza company to place an order, meeting the delivery person, waiting for the pizza to cool down a bit, and then finally eating it, the method might return after each of the first three parts, but it won't complete until the pizza is eaten.

From the developer's point of view, this feels like the method is paused while the asynchronous operation completes. The compiler makes sure that all the local variables used within the method have the same values as they did before the continuation, as it does with iterator blocks.

Let's look at an example of the two cases with a small console application that uses a single asynchronous method awaiting two tasks. `Task.FromResult` always returns a completed task, whereas `Task.Delay` returns a task that completes after the specified delay.

Listing 5.4 Awaiting completed and noncompleted tasks

```
static void Main()
{
    Task task = DemoCompletedAsync();
    Console.WriteLine("Method returned");
    task.Wait();
    Console.WriteLine("Task completed");
}

static async Task DemoCompletedAsync()
{
    Console.WriteLine("Before first await");
    await Task.FromResult(10);
}
```

The diagram illustrates the flow of execution for the code in Listing 5.4. It shows three main points of interest:

- Calls the async method:** An arrow points to the line `Task task = DemoCompletedAsync();`.
- Blocks until the task completes:** An arrow points to the line `task.Wait();`.
- Awaits a completed task:** An arrow points to the line `await Task.FromResult(10);`.

```

Console.WriteLine("Between awaits");
await Task.Delay(1000);
Console.WriteLine("After second await");
}

```

Awaits a noncompleted task

The output from listing 5.4 is as follows:

```

Before first await
Between awaits
Method returned
After second await
Task completed

```

The important aspects of the ordering are as follows:

- The `async` method doesn't return when awaiting the completed task; the method keeps executing synchronously. That's why you see the first two lines with nothing between.
- The `async` method does return when awaiting the delay task. That's why the third line is `Method returned`, printed in the `Main` method. The `async` method can tell that the operation it's waiting for (the delay task) hasn't completed yet, so it returns to avoid blocking.
- The task returned from the `async` method completes only when the method completes. That's why `Task completed` is printed after `After second await`.

I've attempted to capture the `await` expression flow in figure 5.6, although classic flowcharts weren't really designed with asynchronous behavior in mind.

You could think of the dotted line as being another line coming into the top of the flowchart as an alternative. Note that I'm assuming the target of the `await` expression has a result. If you're awaiting a plain `Task` or something similar, *fetch result* really means check that the operation completed successfully.

It's worth stopping to think briefly about what it means to return from an asynchronous method. Again, two possibilities exist:

- This is the first `await` expression you've had to wait for, so you still have the original caller somewhere in your stack. (Remember that until you really need to wait, the method executes synchronously.)
- You've already awaited something else that hadn't already completed,

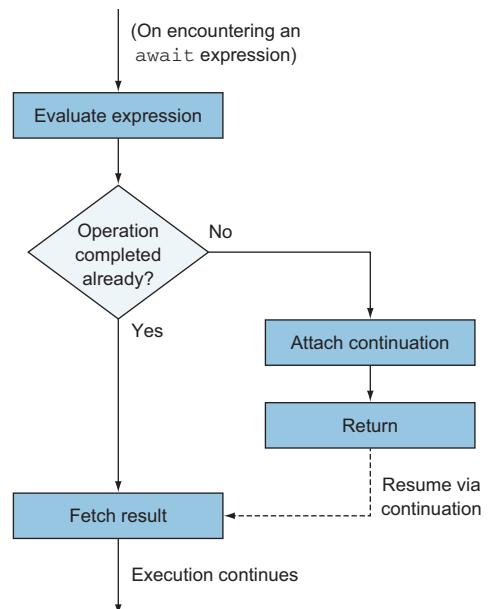


Figure 5.6 User-visible model of await handling

so you’re in a continuation that has been called by *something*. Your call stack will almost certainly have changed significantly from the one you’d have seen when you first entered the method.

In the first case, you’ll usually end up returning a `Task` or `Task<TResult>` to the caller. Obviously, you don’t have the result of the method yet; even if there’s no value to return as such, you don’t know whether the method will complete without exceptions. Because of this, the task you’ll be returning has to be a noncompleted one.

In the latter case, the *something* calling you back depends on your context. For example, in a Windows Forms UI, if you started your async method on the UI thread and didn’t deliberately switch away from it, the whole method would execute on the UI thread. For the first part of the method, you’ll be in some event handler or other—whatever kicked off the async method. Later, however, you’d be called back by the Windows Forms internal machinery (usually known as the *message pump*) pretty directly, as if you were using `Control.BeginInvoke(continuation)`. Here, the calling code—whether it’s the Windows Forms message pump, part of the thread-pool machinery, or something else—doesn’t care about your task.

As a reminder, until you hit the first truly asynchronous `await` expression, the method executes entirely synchronously. Calling an asynchronous method *isn’t* like firing up a new task in a separate thread, and it’s up to you to make sure that you always write async methods so they return quickly. Admittedly, it depends on the context in which you’re writing code, but you should generally avoid performing long-running blocking work in an async method. Separate it out into another method that you can create a `Task` for.

I’d like to briefly revisit the case where the value you’re awaiting is already complete. You might be wondering why an operation that completes immediately would be represented with asynchrony in the first place. It’s a little bit like calling the `Count()` method on a sequence in LINQ: in the general case, you may need to iterate over every item in the sequence, but in some situations (such as when the sequence turns out to be a `List<T>`), an easy optimization is available. It’s useful to have a single abstraction that covers both scenarios, but without paying an execution-time price.

As a real-world example in the asynchronous API case, consider reading asynchronously from a stream associated with a file on disk. All the data you want to read may already have been fetched from disk into memory, perhaps as part of previous `ReadAsync` call request, so it makes sense to use it immediately without going through all the other async machinery. As another example, you may have a cache within your architecture; that can be transparent if you have an asynchronous operation that fetches a value either from the in-memory cache (returning a completed task) or hits storage (returning a noncompleted task that’ll complete when the storage call completes). Now that you know the basics of the flow, you can see where the awaitable pattern fits into the jigsaw.

5.6.3 The use of awaitable pattern members

In section 5.4.1, I described the *awaitable pattern* that a type has to implement in order for you to be able to await an expression of that type. You can now map the different bits of the pattern onto the behavior you’re trying to achieve. Figure 5.7 is the same as figure 5.6 but expanded a little and reworded to use the awaitable pattern instead of general descriptions.

When it’s written like this, you might be wondering what all the fuss is about; why is it worth having language support at all? Attaching a continuation is more complex than you might imagine, though. In simple cases, when the control flow is entirely linear (do some work, await something, do some more work, await something else), it’s pretty easy to imagine what the continuation might look like as a lambda expression, even if it wouldn’t be pleasant. As soon as the code contains loops or conditions, however, and

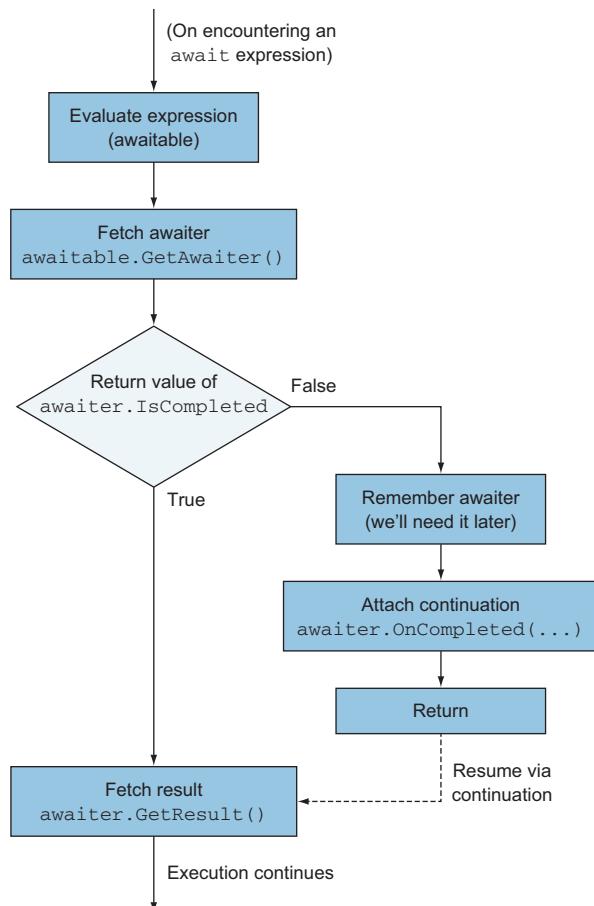


Figure 5.7 Await handling via the awaitable pattern

you want to keep the code within one method, life becomes much more complicated. It's here that the benefits of `async/await` really kick in. Although you could argue that the compiler is merely applying syntactic sugar, there's an enormous difference in readability between manually creating the continuations and getting the compiler to do so for you.

So far, I've described the happy path where all the values we await complete successfully. What happens on failure?

5.6.4 Exception unwrapping

The idiomatic way of representing failures in .NET is via exceptions. Like returning a value to the caller, exception handling requires extra support from the language. When you await an asynchronous operation that's failed, it may have failed a long time ago on a completely different thread. The regular synchronous way of propagating exceptions up the stack doesn't occur naturally. Instead, the `async/await` infrastructure takes steps to make the experience of handling asynchronous failures as similar as possible to synchronous failures. If you think of failure as another kind of result, it makes sense that exceptions and return values are handled similarly. You'll look at how exceptions are propagated out of an asynchronous method in section 5.6.5, but before that, you'll see what happens when you await a failed operation.

In the same way that the `GetResult()` method of an awainer is meant to fetch the return value if there is one, it's also responsible for propagating any exceptions from the asynchronous operation back to the method. This isn't *quite* as simple as it sounds, because in an asynchronous world, a single `Task` can represent multiple operations, leading to multiple failures. Although other awaitable pattern implementations are available, it's worth considering `Task` and `Task<TResult>` specifically, as they're the types you're likely to be awaiting for the vast majority of the time.

`Task` and `Task<TResult>` indicate failures in multiple ways:

- The `Status` of a task becomes `Faulted` when the asynchronous operation has failed (and `IsFaulted` returns `true`).
- The `Exception` property returns an `AggregateException` that contains all the (potentially multiple) exceptions that caused the task to fail or `null` if the task isn't faulted.
- The `Wait()` method throws an `AggregateException` if the task ends up in a faulted state.
- The `Result` property of `Task<TResult>` (which also waits for completion) likewise throws an `AggregateException`.

Additionally, tasks support the idea of cancellation via `CancellationTokenSource` and `CancellationToken`. If a task is canceled, the `Wait()` method and `Result` properties will throw an `AggregateException` containing an `OperationCanceledException` (in practice, a `TaskCanceledException` that derives from `OperationCanceledException`), but the status becomes `Canceled` instead of `Faulted`.

When you await a task, if it's either faulted or canceled, an exception will be thrown but not the `AggregateException`. Instead, for convenience (in most cases), the first exception *within* the `AggregateException` is thrown. In most cases, this is what you want. It's in the spirit of the `async` feature to allow you to write asynchronous code that looks much like the synchronous code you'd otherwise write. For example, consider the following listing, which tries to fetch one URL at a time until either one of them succeeds or you run out of URLs to try.

Listing 5.5 Catching exceptions when fetching web pages

```
async Task<string> FetchFirstSuccessfulAsync(IEnumerable<string> urls)
{
    var client = new HttpClient();
    foreach (string url in urls)
    {
        try
        {
            return await client.GetStringAsync(url);           ↪ Returns the string
                                                               if successful
        }
        catch (HttpRequestException exception)             ↪ Catches and displays
        {
            Console.WriteLine("Failed to fetch {0}: {1}",
                               url, exception.Message);
        }
    }
    throw new HttpRequestException("No URLs succeeded");
}
```

For the moment, ignore the fact that you're losing all the original exceptions and that you're fetching all the pages sequentially. The point I'm trying to make is that catching `HttpRequestException` is what you'd expect here; you're trying an asynchronous operation with an `HttpClient`, and if something fails, it'll throw an `HttpRequestException`. You want to catch and handle that, right? That certainly *feels* like what you'd want to do—but the `GetStringAsync()` call can't throw an `HttpRequestException` for an error such as the server timing out because the method only *starts* the operation. By the time it spots that error, the method has returned. All it can do is return a task that ends up being faulted and containing an `HttpRequestException`. If you simply called `Wait()` on the task, an `AggregateException` would be thrown that contains the `HttpRequestException` within it. The task awainer's `GetResult` method throws the `HttpRequestException` instead, and it's caught by the `catch` block as normal.

Of course, this can lose information. If there are multiple exceptions in a faulted task, `GetResult` can throw only one of them, and it arbitrarily uses the first. You might want to rewrite the preceding code so that on failure, the caller can catch an `AggregateException` and examine *all* the causes of the failure. Importantly, some framework methods do this. For example, `Task.WhenAll()` is a method that'll asynchronously wait for multiple tasks (specified in the method call) to complete. If any of

them fails, the result is a failure that'll contain the exceptions from all the faulted tasks. But if you await only the task returned by `WhenAll()`, you'll see only the first exception. Typically, if you want to check the exceptions in detail, the simplest approach is to use `Task.Exception` for each of the original tasks.

To conclude, you know that theawaiter type's `GetResult()` method is used to propagate both successful results and exceptions when awaiting. In the case of `Task` and `Task<TResult>`, `GetResult()` unwraps a failed task's `AggregateException` to throw the first of its inner exceptions. That explains how an `async` method consumes another asynchronous operation—but how does it propagate its own result to calling code?

5.6.5 Method completion

Let's recap a few points:

- An `async` method usually returns before it completes.
- It returns as soon as it hits an `await` expression where the operation that's being awaited hasn't already finished.
- Assuming it's not a `void` method (in which case the caller has no easy way of telling what's going on), the value the method returns will be a task of some kind: `Task` or `Task<TResult>` before C# 7, with the option of a custom task type (which is explained in section 5.8) in C# 7 and onward. For the moment, let's assume it's a `Task<TResult>` for simplicity.
- That task is responsible for indicating when and how the `async` method completes. If the method completes normally, the task status changes to `RanToCompletion` and the `Result` property holds the return value. If the method body throws an exception, the task status changes to `Faulted` (or `Canceled` depending on the exception) and the exception is wrapped into an `AggregateException` for the task's `Exception` property.
- When the task status changes to any of these terminal states, any continuations associated with it (such as code in any asynchronous method awaiting the task) can be scheduled to run.

Yes, this sounds like it's repetition

You may be wondering whether you've accidentally skipped back a couple of pages and read them twice. Didn't you just look at the same ideas when you awaited something?

Absolutely. All I'm doing is showing what the `async` method does to indicate how it completes rather than how an `await` expression examines how something else has completed. If these didn't feel the same, that would be odd, because usually `async` methods are chained together: the value you're awaiting in one `async` method is probably the value returned by another `async` method. In fancier terms, `async` operations *compose* easily.

All of this is done for you by the compiler with the help of a fair amount of infrastructure. You'll look at some of those details in the next chapter (although not every single nook and cranny; even I have limits). This chapter is more about the behavior you can rely on in your code.

RETURNING SUCCESSFULLY

The success case is the simplest one: if the method is declared to return a `Task<TResult>`, the return statement has to provide a value of type `T` (or something that can be converted to `TResult`), and the `async` infrastructure propagates that to the task.

If the return type is `Task` or `void`, any return statements have to be of the form `return` without a value, or it's fine to let execution reach the end of the method, like a nonasync `void` method. In both cases, there's no value to propagate, but the status of the task changes appropriately.

LAZY EXCEPTIONS AND ARGUMENT VALIDATION

The most important point to note about exceptions is that an `async` method never directly throws an exception. Even if the first thing the method body does is throw an exception, it'll return a faulted task. (The task will be immediately faulted in this case.) This is a bit of a pain in terms of argument validation. Suppose you want to do some work in an `async` method after validating that the parameters don't have null values. If you validate the parameters as you would in a normal synchronous code, the caller won't have any indication of the problem until the task is awaited. The following listing gives an example.

Listing 5.6 Broken argument validation in an `async` method

```
static async Task MainAsync()
{
    Task<int> task = ComputeLengthAsync(null);           ← Deliberately passes
    Console.WriteLine("Fetched the task");
    int length = await task;                            ← Awaits
    Console.WriteLine("Length: {0}", length);            ← the result
}

static async Task<int> ComputeLengthAsync(string text)
{
    if (text == null)
    {
        throw new ArgumentNullException("text");          ← Throws an exception
    }                                                    ← as early as possible
    await Task.Delay(500);                            ← Simulates real
    return text.Length;                             ← asynchronous work
}
```

The output shows `Fetched the task` before it fails. The exception has been thrown synchronously before that output is written, because there are no `await` expressions before the validation, but the calling code won't see it until it awaits the returned task. Some argument validation can sensibly be done up front without taking a long time

(or incurring other asynchronous operations). In these cases, it'd be better if the failure were reported immediately, before the system can get itself into further trouble. As an example, `HttpClient.GetStringAsync` will throw an exception immediately if you pass it a null reference.

NOTE If you've ever written an iterator method that needs to validate its arguments, this may sound familiar. It's not quite the same, but it has a similar effect. In iterator blocks, any code in the method, including argument validation, doesn't execute at all until the first call to `MoveNext()` on the sequence returned by the method. In the asynchronous case, the argument validation occurs immediately, but the exception won't be obvious until you await the result.

You may not be too worried about this. Eager argument validation may be regarded as a nice-to-have feature in many cases. I've certainly become a lot less pedantic about this in my own code, as a matter of pragmatism; in most cases, the difference in timing isn't terribly important. But if you do want to throw an exception synchronously from a method returning a task, you have three options, all of which are variations on the same theme.

The idea is to write a *nonasync* method that returns a task and is implemented by validating the arguments and then calling a separate async function that assumes the argument has already been validated. The three variations are in terms of how the async function is represented:

- You can use a separate async method.
- You can use an async anonymous function (which you'll see in the next section).
- In C# 7 and above, you can use a local async method.

My preference is the last of these; it has the benefit of not introducing another method into the class without the downside of having to create a delegate. Listing 5.7 shows the first option, as that doesn't rely on anything we haven't already covered, but the code for the other options is similar (and is in the downloadable code for book). This is only the `ComputeLengthAsync` method; the calling code doesn't need to change.

Listing 5.7 Eager argument validation with a separate method

```
static Task<int> ComputeLengthAsync(string text)
{
    if (text == null)
    {
        throw new ArgumentNullException("text");
    }
    return ComputeLengthAsyncImpl(text);
}

static async Task<int> ComputeLengthAsyncImpl(string text)
```

Nonasync method so exceptions aren't wrapped in a task

After validation, delegate to implementation method.

```

    await Task.Delay(500);
    return text.Length;
}

```



**Implementation async method
assumes validated input**

Now when `ComputeLengthAsync` is called with a null argument, the exception is thrown synchronously rather than returning a faulted task.

Before moving on to asynchronous anonymous functions, let's briefly revisit cancellation. I've mentioned this a couple of times in passing, but it's worth considering in a bit more detail.

HANDLING CANCELLATION

The Task Parallel Library (TPL) introduced a uniform cancellation model into .NET 4 using two types: `CancellationTokenSource` and `CancellationToken`. The idea is that you can create a `CancellationTokenSource` and then ask it for a `CancellationToken`, which is passed to an asynchronous operation. You can perform the cancellation on only the source, but that's reflected to the token. (Therefore, you can pass out the same token to multiple operations and not worry about them interfering with each other.) There are various ways of using the cancellation token, but the most idiomatic approach is to call `ThrowIfCancellationRequested`, which will throw `OperationCanceledException` if the token has been canceled and will do nothing otherwise.¹ The same exception is thrown by synchronous calls (such as `Task.Wait`) if they're canceled.

How this interacts with asynchronous methods is undocumented in the C# specification. According to the specification, if an asynchronous method body throws *any* exception, the task returned by the method will be in a faulted state. The exact meaning of *faulted* is implementation specific, but in reality, if an asynchronous method throws an `OperationCanceledException` (or a derived exception type, such as `TaskCanceledException`), the returned task will end up with a status of `Canceled`. You can demonstrate that it's only the type of exception that determines the status by throwing an `OperationCanceledException` directly without the use of any cancellation tokens.

Listing 5.8 Creating a canceled task by throwing `OperationCanceledException`

```

static async Task ThrowCancellationException()
{
    throw new OperationCanceledException();
}
...
Task task = ThrowCancellationException();
Console.WriteLine(task.Status);

```

This outputs `Canceled` rather than the `Faulted` you might expect from the specification. If you `Wait()` on the task or ask for its result (in the case of a `Task<TResult>`),

¹ An example of this is available in the downloadable source code.

the exception is still thrown within an `AggregateException`, so it's not like you need to explicitly start checking for cancellation on every task you use.

Off to the races?

You might be wondering if there's a race condition in listing 5.8. After all, you're calling an asynchronous method and then immediately expecting the status to be fixed. If this code were starting a new thread, that would be dangerous—but it's not.

Remember that before the first `await` expression, an asynchronous method runs synchronously. It still performs result and exception wrapping, but the fact that it's in an asynchronous method doesn't necessarily mean there are any more threads involved. The `ThrowCancellationException` method doesn't contain any `await` expressions, so the whole method runs synchronously; you know that we'll have a result by the time it returns. Visual Studio issues a warning for any asynchronous function that doesn't contain any `await` expressions, but in this case it's exactly what you want.

Importantly, if you `await` an operation that's canceled, the original `OperationCanceledException` is thrown. Consequently, unless you take any direct action, the task returned from the asynchronous method will also be canceled; cancellation is propagated in a natural fashion.

Congratulations on making it this far. You've now covered most of the hard parts for this chapter. You still have a couple of features to learn about, but they're much easier to understand than the preceding sections. It'll get tough again in the next chapter when we dissect what the compiler's doing behind the scenes, but for now you can enjoy relative simplicity.

5.7 Asynchronous anonymous functions

I won't spend much time on asynchronous anonymous functions. As you'd probably expect, they're a combination of two features: anonymous functions (lambda expressions and anonymous methods) and asynchronous functions (code that can include `await` expressions). They allow you to create delegates that represent asynchronous operations. Everything you've learned so far about asynchronous methods applies to asynchronous anonymous functions, too.

NOTE In case you were wondering, you can't use asynchronous anonymous functions to create expression trees.

You create an asynchronous anonymous function like any other anonymous method or lambda expression by simply adding the `async` modifier at the start. Here's an example:

```
Func<Task> lambda = async () => await Task.Delay(1000);
Func<Task<int>> anonMethod = async delegate()
{
    Console.WriteLine("Started");
```

```

    await Task.Delay(1000);
    Console.WriteLine("Finished");
    return 10;
};

```

The delegate you create has to have a signature with a return type that would be suitable for an asynchronous method (`void`, `Task`, or `Task<TResult>` for C# 5 and 6, with the option of a custom task type in C# 7). You can capture variables, as with other anonymous functions, and add parameters. Also, the asynchronous operation doesn't start until the delegate is invoked, and multiple invocations create multiple operations. Delegate invocation *does* start the operation, though; as with a call to an `async` method, it's not awaiting the task that starts an operation, and you don't have to use `await` with the result of an asynchronous anonymous function at all. The following listing shows a slightly fuller (although still pointless) example.

Listing 5.9 Creating and calling an asynchronous function using a lambda expression

```

Func<int, Task<int>> function = async x =>
{
    Console.WriteLine("Starting... x={0}", x);
    await Task.Delay(x * 1000);
    Console.WriteLine("Finished... x={0}", x);
    return x * 2;
};
Task<int> first = function(5);
Task<int> second = function(3);
Console.WriteLine("First result: {0}", first.Result);
Console.WriteLine("Second result: {0}", second.Result);

```

I've deliberately chosen the values here so that the second operation completes quicker than the first. But because you're waiting for the first to finish before printing the results (using the `Result` property, which blocks until the task has completed—again, be careful where you run this), the output looks like this:

```

Starting... x=5
Starting... x=3
Finished... x=3
Finished... x=5
First result: 10
Second result: 6

```

All of this behaves exactly the same as if you'd put the asynchronous code into an asynchronous method.

I've written far more `async` methods than `async` anonymous functions, but they can be useful, particularly with LINQ. You can't use them in LINQ query expressions, but calling the equivalent methods directly works. It has limitations, though: because an `async` function can never return `bool`, you can't call `Where` with an `async` function, for example. I've most commonly used `Select` to transform a sequence of tasks of one type to a sequence of tasks of a different type. Now I'll address a feature I've referred to a few times already: an extra level of generalization introduced by C# 7.

5.8 Custom task types in C# 7

In C# 5 and C# 6, asynchronous functions (that is, `async` methods and `async` anonymous functions) could return only `void`, `Task`, or `Task<TResult>`. C# 7 loosens this restriction slightly and allows any type that's decorated in a particular way to be used as a return type for asynchronous functions.

As a reminder, the `async/await` feature has always allowed us to `await` custom types that follow the awaitable pattern. The new feature here permits writing an `async` method that *returns* a custom type.

This is simultaneously complex and simple. It's complex in that if you want to create your own task type, you have some fiddly work ahead of you. It's not for the faint-hearted. It's simple in that you're almost certainly not going to want to do this other than for experimentation; you're going to want to use `ValueTask<TResult>`. Let's look at that now.

5.8.1 The 99.9% case: `ValueTask<TResult>`

At the time of this writing, the `System.Threading.ValueTask<TResult>` type is present out of the box only in the `netcoreapp2.0` framework, but it's also available in the `System.Threading.Tasks.Extensions` package from NuGet, which makes it far more widely applicable. (Most important, that package includes a target for `netstandard1.0`.)

`ValueTask<TResult>` is simple to describe: it's like `Task<TResult>`, but it's a value type. It has an `AsTask` method that allows you to obtain a regular task from it when you want (for example, to include as one element in a `Task.WhenAll` or `Task.WhenAny` call), but most of the time, you'll want to await it as you would a task.

What's the benefit of `ValueTask<TResult>` over `Task<TResult>`? It all comes down to heap allocation and garbage collection. `Task<TResult>` is a class, and although the async infrastructure reuses completed `Task<TResult>` objects in some cases, most `async` methods will need to create a new `Task<TResult>`. Allocating objects in .NET is cheap enough that in many cases you don't need to worry about it, but if you're doing it a lot or if you're working under tight performance constraints, you want to avoid that allocation if possible.

If an `async` method uses an `await` expression on something that's incomplete, object allocation is unavoidable. It'll return immediately, but it has to schedule a continuation to execute the rest of the method when the awaited operation has completed. In most `async` methods, this is the common case; you don't expect the operation you're awaiting to have completed before you await it. In those cases, `ValueTask<TResult>` provides no benefit and can even be a little more expensive.

In a few cases, though, the already completed case is the most common one, and that's where `ValueTask<TResult>` is useful. To demonstrate this, let's consider a simplified version of a real-world example. Suppose you want to read a byte at a time from a `System.IO.Stream` and do so asynchronously. You can easily add a buffering abstraction layer to avoid calling `ReadAsync` on the underlying `Stream` too often, but

you'd then want to add an async method to encapsulate the operation of populate the buffer from the stream where necessary, then return the next byte. You can use `byte?` with a null value to indicate that you've reached the end of the data. That method is easy to write, but if every call to it allocates a new `Task<byte?>`, you'll be hammering the garbage collector pretty hard. With `ValueTask<TResult>`, heap allocation is required only in the rare cases when you need to refill the buffer from the stream. The following listing shows the wrapper type (`ByteStream`) and an example of using it.

Listing 5.10 Wrapping a stream for efficient asynchronous byte-wise access

```

public sealed class ByteStream : IDisposable
{
    private readonly Stream stream;
    private readonly byte[] buffer;
    private int position;
    private int bufferedBytes;

    public ByteStream(Stream stream)
    {
        this.stream = stream;
        buffer = new byte[1024 * 8]; // An 8 KB buffer will mean you rarely need to await.
    }

    public async ValueTask<byte?> ReadByteAsync()
    {
        if (position == bufferedBytes) // Refills the buffer if necessary
        {
            position = 0;
            bufferedBytes = await
                stream.ReadAsync(buffer, 0, buffer.Length) // Asynchronously reads from underlying stream
                .ConfigureAwait(false); // Configures await operation to ignore context
            if (bufferedBytes == 0)
            {
                return null; // Indicates end of stream where appropriate
            }
        }
        return buffer[position++]; // Returns the next byte from the buffer
    }

    public void Dispose()
    {
        stream.Dispose();
    }
}

Sample usage
using (var stream = new ByteStream(File.OpenRead("file.dat")))
{
    while ((nextByte = await stream.ReadByteAsync()).HasValue)
    {
        ConsumeByte(nextByte.Value); // Uses the byte in some way
    }
}

```

For the moment, you can ignore the `ConfigureAwait` call within `ReadByteAsync`. You'll come back to that in section 5.10, when you look at how to use `async/await` effectively. The rest of the code is straightforward, and all of it could be written without `ValueTask<TResult>`; it'd just be much less efficient.

In this case, most invocations of our `ReadByteAsync` method wouldn't even use the `await` operator because you'd still have buffered data to return, but it'd be equally useful if you were awaiting another value that's usually complete immediately. As I explained in section 5.6.2, when you await an operation that's already complete, the execution continues synchronously, which means you don't need to schedule a continuation and can avoid object allocations.

This is a simplified version of a prototype of the `CodedInputStream` class from the `Google.Protobuf` package, the .NET implementation of Google's Protocol Buffers serialization protocol. In reality, there are multiple methods, each reading a small amount of data either synchronously or asynchronously. Deserializing a message with lots of integer fields can involve a lot of method calls, and making the asynchronous methods return a `Task<TResult>` each time would've been prohibitively inefficient.

NOTE You may be wondering what to do if you have an `async` method that doesn't return a value (so would normally have a return type of `Task`), but that still falls into the category of completing without having to schedule any continuations. In this case, you can stick to returning `Task`: the `async/await` infrastructure caches a task that it can return from any `async` method declared to return `Task` that completes synchronously and without an exception. If the method completes synchronously but with an exception, the cost of allocating a `Task` object likely will be dwarfed by the exception overhead anyway.

For most of us, the ability to use `ValueTask<TResult>` as a return type for `async` methods is the real benefit of C# 7 in terms of asynchrony. But this has been implemented in a general-purpose way, allowing you to create your own return types for `async` methods.

5.8.2 The 0.1% case: Building your own custom task type

I'd like to emphasize again that you're almost certainly never going to need this information. I'm not going to even try to provide a use case beyond `ValueTask<TResult>`, because anything I could think of would be obscure. That said, this book would be incomplete if I didn't show the pattern the compiler uses to determine that a type is a task type. I'll show the details of how the compiler uses the pattern in the next chapter, when you look at the code that gets generated for an `async` method.

Obviously, a custom task type has to implement the awaitable pattern, but there's much more to it than that. To create a custom task type, you have to write a corresponding *builder type* and use the `System.Runtime.CompilerServices.AsyncMethodBuilderAttribute` to let the compiler know the relationship between the two types. This is a new attribute available in the same NuGet package as `ValueTask<TResult>`,

but if you don't want the extra dependency, you can include your own declaration of the attribute (in the right namespace and with the appropriate `BuilderType` property). The compiler will then accept that as a way of decorating task types.

The task type can be generic in a single type parameter or nongeneric. If it's generic, that type parameter must be the type of `GetResult` in the awaiter type; if it's nongeneric, `GetResult` must have a `void` return type.² The builder must be generic or nongeneric in the same way as the task type.

The builder type is the part where the compiler interacts with your code when it's compiling a method returning your custom type. It needs to know how to create your custom task, propagate completion or exceptions, resume after a continuation, and so on. The set of methods and properties you need to provide is significantly more complex than the awaitable pattern. It's easiest to show a complete example, in terms of the members you need to provide, without any implementation.

Listing 5.11 Skeleton of the members required for a generic task type

```
[AsyncMethodBuilder(typeof(CustomTaskBuilder<>))]
public class CustomTask<T>
{
    public CustomTaskAwaiter<T> GetAwaiter();
}

public class CustomTaskAwaiter<T> : INotifyCompletion
{
    public bool IsCompleted { get; }
    public T GetResult();
    public void OnCompleted(Action continuation);
}

public class CustomTaskBuilder<T>
{
    public static CustomTaskBuilder<T> Create();

    public void Start<TStateMachine>(ref TStateMachine stateMachine)
        where TStateMachine : IAsyncStateMachine;

    public void SetStateMachine(IAsyncStateMachine stateMachine);
    public void SetException(Exception exception);
    public void SetResult(T result);

    public void AwaitOnCompleted<TAwaiter, TStateMachine>
        (ref TAwaiter awainer, ref TStateMachine stateMachine)
        where TAwaiter : INotifyCompletion
        where TStateMachine : IAsyncStateMachine;

    public void AwaitUnsafeOnCompleted<TAwaiter, TStateMachine>
        (ref TAwaiter awainer, ref TStateMachine stateMachine)
```

² This surprised me somewhat. It means you can't write a custom task type that always represents an operation returning a string, for example. Given how niche the whole feature is, the likelihood of anyone really wanting a niche use case within the feature is pretty small.

```

    where TAwaiter : INotifyCompletion
    where TStateMachine : IAsyncStateMachine;

    public CustomTask<T> Task { get; }
}

```

This code shows a generic custom task type. For a nongeneric type, the only difference in the builder would be that `SetResult` would be a parameterless method.

One interesting requirement is the `AwaitUnsafeOnCompleted` method. As you'll see in the next chapter, the compiler has the notion of *safe awaiting* and *unsafe awaiting*, where the latter relies on the awaitable type to handle context propagation. A custom task builder type has to handle resuming from both kind of awaiting.

NOTE The term *unsafe* here isn't directly related to the `unsafe` keyword, although similarities exist in terms of "here be dragons, take care!"

To reiterate one final time, you almost certainly don't want to be doing this except as a matter of interest. I don't expect to ever implement my own task type for production code, but I'll certainly use `ValueTask<TResult>`, so I'm still grateful that the feature exists.

Speaking of useful new features, C# 7.1 has one additional feature to mention. Fortunately, it's considerably simpler than custom task types.

5.9 Async main methods in C# 7.1

The requirements for the entry point have remained the same in C# for a long time:

- It must be a method called `Main`.
- It must be static.
- It must have a `void` or `int` return type.
- It must either be parameterless or have a single (non-`ref`, non-`out`) parameter of type `string[]`.
- It must be nongeneric and declared in a nongeneric type (including any containing types being nongeneric, if it's declared in a nested type).
- It can't be a partial method without implementation.
- It can't have the `async` modifier.

With C# 7.1, the final requirement has been dropped but with a slightly different requirement around the return type. In C# 7.1, you can write an `async` entry point (still called `Main`, not `MainAsync`), but it has to have a return type of either `Task` or `Task<int>` corresponding to a synchronous return type of `void` or `int`. Unlike most `async` methods, an `async` entry point can't have a return type of `void` or use a custom task type.

Beyond that, it's a regular `async` method. For example, the following listing shows an `async` entry point that prints two lines to the console with a delay between them.

Listing 5.12 A simple async entry point

```
static async Task Main()
{
    Console.WriteLine("Before delay");
    await Task.Delay(1000);
    Console.WriteLine("After delay");
}
```

The compiler handles async entry points by creating a synchronous wrapper method that it marks as the real entry point into the assembly. The wrapper method is either parameterless or has a `string[]` parameter and either returns `void` or `int`, depending on what the async entry point has in terms of parameters and return type. The wrapper method calls the real code and then calls `GetAwaiter()` on the returned task and `GetResult()` on the awainer. For example, the wrapper method generated for listing 5.11 would look something like this:

```
static void <Main>()
{
    Main().GetAwaiter().GetResult();
```



Method has a name that's invalid in C# but valid in IL.

Async entry points are handy for writing small tools or exploratory code that uses an async-oriented API such as Roslyn.

Those are all the async features from a language perspective. But knowing the capabilities of the language is different from knowing how to use those capabilities effectively. That's particularly true for asynchrony, which is an inherently complex topic.

5.10 Usage tips

This section could never be a complete guide to using asynchrony effectively; that could fill an entire book on its own. We're coming to the end of a chapter that's already long, so I've restrained myself to offer just the most important tips in my experience. I strongly encourage you to read the perspectives of other developers. In particular, Stephen Cleary and Stephen Toub have written reams of blog posts and articles that go into many aspects in great depth. In no particular order, this section provides the most useful suggestions I can make reasonably concisely.

5.10.1 Avoid context capture by using `ConfigureAwait` (where appropriate)

In sections 5.2.2 and 5.6.2, I described synchronization contexts and their effect on the `await` operator. For example, if you're running on a UI thread in WPF or WinForms and you await an asynchronous operation, the UI synchronization context and the async infrastructure make sure that the continuation that runs *after* the `await` operator still runs on that same UI thread. That's exactly what you want in UI code, because you can then safely access the UI afterward.

But when you’re writing library code—or code in an application that doesn’t touch the UI—you don’t want to come back to the UI thread, even if you were originally running in it. In general, the less code that executes in the UI thread, the better. This allows the UI to update more smoothly and avoids the UI thread being a bottleneck. Of course, if you’re writing a UI library, you probably do want to return to the UI thread, but most libraries—for business logic, web services, database access and the like—don’t need this.

The `ConfigureAwait` method is designed precisely for this purpose. It takes a parameter that determines whether the returned awaitable will capture the context when it’s awaited. In practice, I think I’ve always seen the value `false` passed in as an argument. In library code, you wouldn’t write the page-length-fetching code as you saw it earlier:

```
static async Task<int> GetPageLengthAsync(string url)
{
    var fetchTextTask = client.GetStringAsync(url);
    int length = (await fetchTextTask).Length;
    return length;
} ← Imagine more  
code here
```

Instead, you’d call `ConfigureAwait(false)` on the task returned by `client.GetStringAsync(url)` and await the result:

```
static async Task<int> GetPageLengthAsync(string url)
{
    var fetchTextTask = client.GetStringAsync(url).ConfigureAwait(false);
    int length = (await fetchTextTask).Length;
    return length;
} ← Same additional  
code
```

I’ve cheated a little here by using implicit typing for the `fetchTextTask` variable. In the first example, it’s a `Task<int>`; in the second, it’s a `ConfiguredTaskAwaitable<int>`. Most code I’ve seen awaits the result directly anyway, though, like this:

```
string text = await client.GetStringAsync(url).ConfigureAwait(false);
```

The result of calling `ConfigureAwait(false)` is that the continuation won’t be scheduled against the original synchronization context; it’ll execute on a thread-pool thread. Note that the behavior differs from the original code only if the task hasn’t already completed by the time it’s awaited. If it has already completed, the method continues executing synchronously, even in the face of `ConfigureAwait(false)`. Therefore, every task you await in a library should be configured like this. You can’t just call `ConfigureAwait(false)` on the first task in an `async` method and rely on the rest of the method executing on a thread-pool thread.

All of this means you need to be careful when writing library code. I expect that eventually a better solution may exist (setting the default for a whole assembly, for example), but for the moment, you need to be vigilant. I recommend using a Roslyn analyzer to spot where you've forgotten to configure a task before awaiting it. I've had positive experiences with the `ConfigureAwaitChecker` NuGet package, but others are available, too.

In case you're worried about what this does to the caller, you don't need to be. Suppose the caller is awaiting the task returned by `GetPageLengthAsync` and then updating a user interface to display the result. Even if the continuation within `GetPageLengthAsync` runs on a thread-pool thread, the `await` expression performed in the UI code will capture the UI context and schedule *its* continuation to run on the UI thread, so the UI can still be updated after that.

5.10.2 Enable parallelism by starting multiple independent tasks

In section 5.6.1, you looked at multiple pieces of code to achieve the same goal: find out how much to pay an employee based on their hourly rate and how many hours they'd worked. The last two pieces of code were like this:

```
Task<decimal> hourlyRateTask = employee.GetHourlyRateAsync();
decimal hourlyRate = await hourlyRateTask;
Task<int> hoursWorkedTask = timeSheet.GetHoursWorkedAsync(employee.Id);
int hoursWorked = await hoursWorkedTask;
AddPayment(hourlyRate * hoursWorked);
```

and this

```
Task<decimal> hourlyRateTask = employee.GetHourlyRateAsync();
Task<int> hoursWorkedTask = timeSheet.GetHoursWorkedAsync(employee.Id);
AddPayment(await hourlyRateTask * await hoursWorkedTask);
```

In addition to being shorter, the second piece of code introduces parallelism. Both tasks can be started independently, because you don't need the output of the second task as input into the first task. This doesn't mean that the async infrastructure creates any more threads. For example, if the two asynchronous operations here are web services, both requests to the web services can be in flight without any threads being blocked on the result.

The shortness aspect is only incidental here. If you want the parallelism but like having the separate variables, that's fine:

```
Task<decimal> hourlyRateTask = employee.GetHourlyRateAsync();
Task<int> hoursWorkedTask = timeSheet.GetHoursWorkedAsync(employee.Id);
decimal hourlyRate = await hourlyRateTask;
int hoursWorked = await hoursWorkedTask;
AddPayment(hourlyRate * hoursWorked);
```

The only difference between this and the original code is that I swapped the second and third lines. Instead of awaiting `hourlyRateTask` and then starting `hoursWorkedTask`, you start both tasks and then await both tasks.

In most cases, if you can perform independent work in parallel, it's a good idea to do so. Be aware that if `hourlyRateTask` fails, you won't observe the result of `hoursWorkedTask`, including any failures in that task. If you need to log all task failures, for example, you might want to use `Task.WhenAll` instead.

Of course, this sort of parallelization relies on the tasks being independent to start with. In some cases, the dependency may not be entirely obvious. If you have one task that's authenticating a user and another task performing an action on their behalf, you'd want to wait until you'd checked the authentication before starting the action, even if you *could* write the code to execute in parallel. The `async/await` feature can't make these decisions for you, but it makes it easy to parallelize asynchronous operations when you've decided that it's appropriate.

5.10.3 Avoid mixing synchronous and asynchronous code

Although asynchrony isn't entirely all or nothing, it gets much harder to implement correctly when some of your code is synchronous and other parts are asynchronous. Switching between the two approaches is fraught with difficulties—some subtle, others less so. If you have a network library that exposes only synchronous operations, writing an asynchronous wrapper for those operations is difficult to do safely, and likewise in reverse.

In particular, be aware of the dangers of using the `Task<TResult>.Result` property and `Task.Wait()` methods to try to synchronously retrieve the result of an asynchronous operation. This can easily lead to deadlock. In the most common case, the asynchronous operation requires a continuation to execute in a thread that's blocked, waiting for the operation to compete.

Stephen Toub has a pair of excellent and detailed blog posts on this topic: “Should I expose synchronous wrappers for asynchronous methods?” and “Should I expose asynchronous wrappers for synchronous methods?” (Spoiler alert: the answer is no in both cases, as you've probably guessed.) As with all rules, there are exceptions, but I strongly advise that you make sure you thoroughly understand the rule before breaking it.

5.10.4 Allow cancellation wherever possible

Cancellation is one area that doesn't have a strong equivalent in synchronous code, where you usually have to wait for a method to return before continuing. The ability to cancel an asynchronous operation is extremely powerful, but it does rely on cooperation throughout the stack. If you want to use a method that doesn't allow you to pass in a cancellation token, there's not an awful lot you can do about it. You can write somewhat-intricate code so that your `async` method completes with a canceled status and ignore the final result of the noncancelable task, but that's far from ideal. You really want to be able to stop any work in progress, and you also don't want to have to worry about any disposable resources that could be returned by the asynchronous method when it does eventually complete.

Fortunately, most low-level asynchronous APIs do expose a cancellation token as a parameter. All you need to do is follow the same pattern yourself, typically passing the same cancellation token you receive in the parameter as an argument to all asynchronous methods you call. Even if you don't currently have any requirements to allow cancellation, I advise providing the option consistently right from the start, because it's painful to add later.

Again, Stephen Toub has an excellent blog post on the subtle difficulties of trying to work around noncancelable asynchronous operations. Search for "How do I cancel non-cancelable async operations?" to find it.

5.10.5 Testing `asynchrony`

Testing asynchronous code can be extremely tricky, particularly if you want to test the asynchrony itself. (Tests that answer questions such as "What happens if I cancel the operation between the second and third asynchronous calls within the method?" require quite elaborate work.)

It's not impossible, but be prepared for an uphill battle if you want to test comprehensively. When I wrote the third edition of this book, I hoped that by 2019 there would be robust frameworks to make all of this relatively simple. Unfortunately, I'm disappointed.

Most unit-test frameworks do have support for asynchronous tests, however. That support is pretty much vital to write tests for asynchronous methods, for all the reasons I mentioned before about the difficulties in mixing synchronous and asynchronous code. Typically, writing an asynchronous test is as simple as writing a test method with the `async` modifier and declaring it to return `Task` instead of `void`:

```
[Test]
public async Task FooAsync()
{
    ← Code to test your FooAsync
        production method
}
```

Test frameworks often provide an `Assert.ThrowsAsync` method for testing that a call to an asynchronous method returns a task that eventually becomes faulted.

When testing asynchronous code, often you'll want to create a task that's already completed, with a particular result or fault. The methods `Task.FromResult`, `Task.FromException`, and `Task.FromCanceled` are useful here.

For more flexibility, you can use `TaskCompletionSource<TResult>`. This type is used by a lot of the `async` infrastructure in the framework. It effectively allows you to create a task representing an ongoing operation and then set the result (including any exception or cancellation) later, at which point the task will complete. This is extremely useful when you want to return a task from a mocked dependency but make that returned task complete later in the test.

One aspect of `TaskCompletionSource<TResult>` to know about is that when you set the result, continuations attached to the associated task can run synchronously

on the same thread. The exact details of how the continuations are run depend on various aspects of the threads and synchronization contexts involved, and after you’re aware of it as a possibility, it’s relatively easy to take account of. Now you’re aware and can hopefully avoid wasting the time being baffled in the way that I was.

This is an incomplete summary of what I’ve learned over the last four years or so of writing asynchronous code, but I don’t want to lose sight of the topic of the book (the C# language, not asynchrony). You’ve seen what the `async/await` feature does, from the developer’s perspective. You haven’t looked at what happens under the hood in any detail yet, although the awaitable pattern provides some clues.

If you haven’t played with `async/await` yet, I *strongly* advise that you do so now, before taking on the next chapter, which looks at the implementation details. Those details are important but are tricky to understand at the best of times and will be hard to understand if you don’t have some experience of using `async/await`. If you don’t have that experience yet and don’t particularly want to put the time in right now, I advise skipping the next chapter for now. It’s only about the implementation details of asynchrony; I promise you won’t miss anything else.

Summary

- The core of asynchrony is about starting an operation and then later continuing when the operation has completed without having to block in the middle.
- `Async/await` allows you to write familiar-looking code that acts asynchronously.
- `Async/await` handles synchronization contexts so UI code can start an asynchronous operation and then continue on the UI thread when that operation has finished.
- Successful results and exceptions are propagated through asynchronous operations.
- Restrictions limit where you can use the `await` operator, but C# 6 (and later) versions have fewer restrictions than C# 5.
- The compiler uses the awaitable pattern to determine which types can be awaited.
- C# 7 allows you to create your own custom task type, but you almost certainly want to use `ValueTask<TResult>`.
- C# 7.1 allows you to write `async Main` methods as program entry points.

Async implementation



This chapter covers

- The structure of asynchronous code
- Interacting with the framework builder types
- Performing a single step in an async method
- Understanding execution context flow across await expressions
- Interacting with custom task types

I vividly remember the evening of October 28, 2010. Anders Hejlsberg was presenting `async/await` at PDC, and shortly before his talk started, an avalanche of downloadable material was made available, including a draft of the changes to the C# specification, a Community Technology Preview (CTP) of the C# 5 compiler, and the slides Anders was presenting. At one point, I was watching the talk live and skimming through the slides while the CTP installed. By the time Anders had finished, I was writing `async` code and trying things out.

In the next few weeks, I started taking bits apart and looking at exactly what code the compiler was generating, trying to write my own simplistic implementation of

the library that came with the CTP, and generally poking at it from every angle. As new versions came out, I worked out what had changed and became more and more comfortable with what was going on behind the scenes. The more I saw, the more I appreciated how much boilerplate code the compiler is happy to write on our behalf. It's like looking at a beautiful flower under a microscope: the beauty is still there to be admired, but there's so much more to it than can be seen at first glance.

Not everyone is like me, of course. If you just want to rely on the behavior I've already described and simply trust that the compiler will do the right thing, that's absolutely fine. Alternatively, you won't miss out on anything if you skip this chapter for now and come back to it at a later date; none of the rest of the book relies on it. It's unlikely that you'll ever have to debug your code down to the level that you'll look at here, but I believe this chapter will give you more insight into how `async/await` hangs together. Both the awaitable pattern and the requirements for custom task types make more sense after you've looked at the generated code. I don't want to get too mystical about this, but there's a certain connection between the language and the developer that's enriched by studying these implementation details.

As a rough approximation, we'll pretend that the C# compiler performs a transformation from C# code using `async/await` to C# code without using `async/await`. Of course, the compiler is able to operate at a lower level than this with intermediate representations that can be emitted as IL. Indeed, in some aspects of `async/await`, the IL generated can't be represented in regular C#, but it's easy enough to explain those places.

Debug and release builds differ, and future implementations may, too

While writing this chapter, I became aware of a difference between debug and release builds of async code: in debug builds, the generated state machines are classes rather than structs. (This is to give a better debugger experience; in particular, it gives more flexibility in Edit and Continue scenarios.) This wasn't true when I wrote the third edition; the compiler implementation has changed. It may change again in the future, too. If you decompile async code compiled by a C# 8 compiler, it could look slightly different from what's presented here.

Although this is surprising, it shouldn't be too alarming. By definition, implementation details can change over time. None of this invalidates any of the insight to be gained from studying a particular implementation. Just be aware that this is a different kind of learning from "these are the rules of C#, and they'll change only in well-specified ways."

In this chapter, I show the code generated by a release build. The differences mostly affect performance, and I believe most readers will be more interested in the performance of release builds than debug builds.

The generated code is somewhat like an onion; it has layers of complexity. We'll start from the very outside and work our way in toward the tricky bit: `await` expressions and

the dance of awaiters and continuations. For the sake of brevity, I'm going to present only asynchronous methods, not `async` anonymous functions; the machinery between the two is the same anyway, so there's nothing particularly interesting to learn by repeating the work.

6.1 Structure of the generated code

As I mentioned in chapter 5, the implementation (both in this approximation and in the code generated by the real compiler) is in the form of a *state machine*. The compiler will generate a private nested struct to represent the asynchronous method, and it must also include a method with the same signature as the one you've declared. I call this the *stub method*; there's not much to it, but it starts all of the rest going.

NOTE Frequently, I'm going to talk about the state machine *pausing*. This corresponds to a point where the `async` method reaches an `await` expression and the operation being awaited hasn't completed yet. As you may remember from chapter 5, when that happens, a continuation is scheduled to execute the rest of the `async` method when the awaited operation has completed, and then the `async` method returns. Similarly, it's useful to talk about the `async` method taking a *step*: the code it executes between pauses, effectively. These aren't official terms, but they're useful as shorthand.

The state machine keeps track of where you are within the `async` method. Logically, there are four kinds of state, in common execution order:

- Not started
- Executing
- Paused
- Complete (either successfully or faulted)

Only the Paused set of states depends on the structure of the `async` method. Each `await` expression within the method is a distinct state to be returned to in order to trigger more execution. While the state machine is executing, it doesn't need to keep track of the exact piece of code that's executing; at that point, it's just regular code, and the CPU keeps track of the instruction pointer just as with synchronous code. The state is recorded when the state machine needs to pause; the whole purpose is to allow it to continue the code execution later from the point it reached. Figure 6.1 shows the transitions between the possible states.

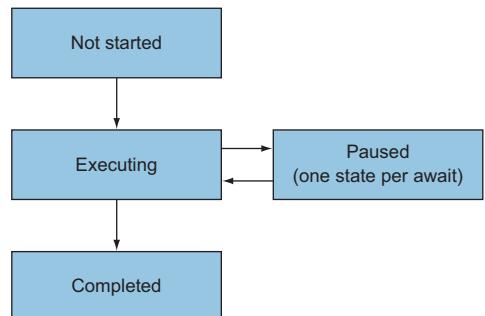


Figure 6.1 State transition diagram

Let's make this concrete with a real piece of code. The following listing shows a simple `async` method. It's not quite as simple as you could make it, but it can demonstrate a few things at the same time.

Listing 6.1 Simple introductory `async` method

```
static async Task PrintAndWait(TimeSpan delay)
{
    Console.WriteLine("Before first delay");
    await Task.Delay(delay);
    Console.WriteLine("Between delays");
    await Task.Delay(delay);
    Console.WriteLine("After second delay");
}
```

Three points to note at this stage are as follows:

- You have a parameter that you'll need to use in the state machine.
- The method includes two `await` expressions.
- The method returns `Task`, so you need to return a task that will complete after the final line is printed, but there's no specific result.

This is nice and simple because you have no loops or `try/catch/finally` blocks to worry about. The control flow is simple, apart from the awaiting, of course. Let's see what the compiler generates for this code.

Do try this at home

I typically use a mixture of `ildasm` and Redgate Reflector for this sort of work, setting the Optimization level to C# 1 to prevent the decompiler from reconstructing the `async` method for us. Other decompilers are available, but whichever one you pick, I recommend checking the IL as well. I've seen subtle bugs in decompilers when it comes to `await`, often in terms of the execution order.

You don't have to do any of this if you don't want to, but if you find yourself wondering what the compiler does with a particular code construct, and this chapter doesn't provide the answer, just go for it. Don't forget the difference between debug and release builds, though, and don't be put off by the names generated by the compiler, which can make the result harder to read.

Using the tools available, you can decompile listing 6.1 into something like listing 6.2. Many of the names that the C# compiler generates aren't valid C#, I've rewritten them as valid identifiers for the sake of getting runnable code. In other cases, I've renamed the identifiers to make the code more readable. Later, I've taken a few liberties with how the cases and labels for the state machine are ordered; it's absolutely logically equivalent to the generated code, but much easier to read. In other places, I've used a

switch statement even with only two cases, where the compiler might effectively use if/else. In these places, the switch statement represents the more general case that can work when there are multiple points to jump to, but the compiler can generate simpler code for simpler situations.

Listing 6.2 Generated code for listing 6.1 (except for MoveNext)
Stub method

```
[AsyncStateMachine(typeof(PrintAndWaitStateMachine))]
[DebuggerStepThrough]
private static unsafe Task PrintAndWait(TimeSpan delay)
{
    var machine = new PrintAndWaitStateMachine
    {
        delay = delay,
        builder = AsyncTaskMethodBuilder.Create(),
        state = -1
    };
    machine.builder.Start(ref machine);
    return machine.builder.Task;
}
```

The diagram shows the Stub method with several annotations:

- Original method parameter**: Points to the `delay` parameter.
- Initializes the state machine, including method parameters**: Points to the line `var machine = new PrintAndWaitStateMachine`.
- Returns the task representing the async operation**: Points to the line `return machine.builder.Task;`
- Runs the state machine until it needs to wait**: Points to the `builder.Start` call.

Private struct for the state machine

```
[CompilerGenerated]
private struct PrintAndWaitStateMachine : IAsyncStateMachine
{
    public int state;
    public AsyncTaskMethodBuilder builder;
    private TaskAwaiter awariter;
    public TimeSpan delay;

    void IAsyncStateMachine.MoveNext()
    {
    }

    [DebuggerHidden]
    void IAsyncStateMachine.SetStateMachine(
        IAsyncStateMachine stateMachine)
    {
        this.builder.SetStateMachine(stateMachine);
    }
}
```

The diagram shows the Private struct with several annotations:

- Original method parameter**: Points to the `delay` parameter.
- State of the state machine (where to resume)**: Points to the `state` field.
- The builder hooking into async infrastructure types**: Points to the `builder` field.
- Awariter to fetch result from when resuming**: Points to the `awariter` field.
- Main state machine work goes here.**: Points to the `MoveNext` and `SetStateMachine` methods.
- Connects the builder and the boxed state machine**: Points to the `builder.SetStateMachine` call.

This listing looks somewhat complicated already, but I should warn you that the bulk of the work is done in the `MoveNext` method, and I've completely removed the implementation of that for now. The point of listing 6.2 is to set the scene and provide the structure so that when you get to the `MoveNext` implementation, it makes sense. Let's look at the pieces of the listing in turn, starting with the stub method.

6.1.1 The stub method: Preparation and taking the first step

The stub method from listing 6.2 is simple apart from the `AsyncTaskMethodBuilder`. This is a value type, and it's part of the common async infrastructure. You'll see over the rest of the chapter how the state machine interacts with the builder.

```
[AsyncStateMachine(typeof(PrintAndWaitStateMachine))]  
[DebuggerStepThrough]  
private static unsafe Task PrintAndWait(TimeSpan delay)  
{  
    var machine = new PrintAndWaitStateMachine  
    {  
        delay = delay,  
        builder = AsyncTaskMethodBuilder.Create(),  
        state = -1  
    };  
    machine.builder.Start(ref machine);  
    return machine.builder.Task;  
}
```

The attributes applied to the method are essentially for tooling. They have no effect on regular execution, and you don't need to know any details about them in order to understand the generated asynchronous code. The state machine is always created in the stub method with three pieces of information:

- Any parameters (in this case, just `delay`), each as separate fields in the state machine
- The builder, which varies depending on the return type of the async method
- The initial state, which is always `-1`

NOTE The name `AsyncTaskMethodBuilder` may make you think of reflection, but it's not creating a method in IL or anything like that. The builder provides functionality that the generated code uses to propagate success and failure, handle awaiting, and so forth. If the name "helper" works better for you, feel free to think of it that way.

After creating the state machine, the stub method asks the machine's builder to start it, passing the machine itself by reference. You'll see quite a lot of passing by reference in the following few pages, and this comes down to a need for efficiency and consistency. Both the state machine and the `AsyncTaskMethodBuilder` are *mutable* value types. Passing `machine` by reference to the `Start` method avoids making a copy of the state, which is more efficient and ensures that any changes made to the state within `Start` are still visible when the `Start` method returns. In particular, the `builder` state within the machine may well change during `Start`. That's why it's important that you use `machine.builder` for both the `Start` call and the `Task` property afterward. Suppose you extracted `machine.builder` to a local variable, like this:

```
var builder = machine.builder;  
builder.Start(ref machine);  
return builder.Task;
```

Invalid attempt
at refactoring

With that code, state changes made directly within `builder.Start()` wouldn't be seen within `machine.builder` (or vice versa) because it would be a copy of the builder. This is where it's important that `machine.builder` refers to a field, not a property. You don't want to operate on a copy of the builder in the state machine; rather, you want to operate directly on the value that the state machine contains. This is precisely the sort of detail that you don't want to have to deal with yourself and is why mutable value types and public fields are almost always a bad idea. (You'll see in chapter 11 how they can be useful when carefully considered.)

Starting the machine doesn't create any new threads. It just runs the state machine's `MoveNext()` method until either the state machine needs to pause while it awaits another asynchronous operation or completes. In other words, it takes one step. Either way, `MoveNext()` returns, at which point `machine.builder.Start()` returns, and you can return a task representing the overall asynchronous method back to our caller. The builder is responsible for creating the task and ensuring that it changes state appropriately over the course of the asynchronous method.

That's the stub method. Now let's look at the state machine itself.

6.1.2 Structure of the state machine

I'm still omitting the majority of the code from the state machine (in the `MoveNext()` method), but here's a reminder of the structure of the type:

```
[CompilerGenerated]
private struct PrintAndWaitStateMachine : IAsyncStateMachine
{
    public int state;
    public AsyncTaskMethodBuilder builder;
    private TaskAwaiter awaiter;
    public TimeSpan delay;

    void IAsyncStateMachine.MoveNext()
    {
        Implementation
        omitted
    }

    [DebuggerHidden]
    void IAsyncStateMachine.SetStateMachine(
        IAsyncStateMachine stateMachine)
    {
        this.builder.SetStateMachine(stateMachine);
    }
}
```

Again, the attributes aren't important. The important aspects of the type are as follows:

- It implements the `IAsyncStateMachine` interface, which is used for the async infrastructure. The interface has only the two methods shown.
- The fields, which store the information the state machine needs to remember between one step and the next.

- The `MoveNext()` method, which is called once when the state machine is started and once each time it resumes after being paused.
- The `SetStateMachine()` method, which always has the same implementation (in release builds).

You've seen one use of the type implementing `IAsyncStateMachine` already, although it was somewhat hidden: `AsyncTaskMethodBuilder.Start()` is a generic method with a constraint that the type parameter has to implement `IAsyncStateMachine`. After performing a bit of housekeeping, `Start()` calls `MoveNext()` to make the state machine take the first step of the async method.

The fields involved can be broadly split into five categories:

- The current state (for example, not started, paused at a particular await expression, and so forth)
- The method builder used to communicate with the async infrastructure and to provide the Task to return
- Awaiters
- Parameters and local variables
- Temporary stack variables

The state and builder are fairly simple. The state is just an integer with one of the following values:

- `-1`—Not started, or currently executing (it doesn't matter which)
- `-2`—Finished (either successfully or faulted)
- *Anything else*—Paused at a particular await expression

As I mentioned before, the type of the builder depends on the return type of the async method. Before C# 7, the builder type was always `AsyncVoidMethodBuilder`, `AsyncTaskMethodBuilder`, or `AsyncTaskMethodBuilder<T>`. With C# 7 and custom task types, the builder type specified by the `AsyncTaskMethodBuilderAttribute` is applied to the custom task type.

The other fields are slightly trickier in that all of them depend on the body of the async method, and the compiler tries to use as few fields as it can. The crucial point to remember is that you need fields only for values that you need to come back to after the state machine resumes at some point. Sometimes the compiler can use fields for multiple purposes, and sometimes it can omit them entirely.

The first example of how the compiler can reuse fields is with awaiters. Only one awainer is relevant at a time, because any particular state machine can await only one value at a time. The compiler creates a single field for each awainer type that's used. If you await two `Task<int>` values, one `Task<string>`, and three nongeneric `Task` values in an async method, you'll end up with three fields: a `TaskAwainer<int>`, a `TaskAwainer<string>`, and a nongeneric `TaskAwainer`. The compiler uses the appropriate field for each await expression based on the awainer type.

NOTE This assumes the awainer is introduced by the compiler. If you call `GetAwaiter()` yourself and assign the result to a local variable, that's treated like any other local variable. I'm talking about the awainers that are produced as the result of await expressions.

Next, let's consider local variables. Here, the compiler doesn't reuse fields but can omit them entirely. If a local variable is used only between two await expressions rather than *across* await expressions, it can stay as a local variable in the `MoveNext()` method.

It's easier to see what I mean with an example. Consider the following async method:

```
public async Task LocalVariableDemoAsync()
{
    int x = DateTime.UtcNow.Second;           ← x is assigned
    int y = DateTime.UtcNow.Second;           | y is used only
    Console.WriteLine(y);                   | before the await.
    await Task.Delay();                     |
    Console.WriteLine(x);                  ← x is used after
}                                         | the await.
```

The compiler would generate a field for `x` because the value has to be preserved while the state machine is paused, but `y` can just be a local variable on the stack while the code is executing.

NOTE The compiler does a pretty good job of creating only as many fields as it needs. But at times, you might spot an optimization that the compiler could perform but doesn't. For example, if two variables have the same type and are both used across await expressions (so they need fields), but they're never both in scope at the same time, the compiler could use just one field for both as it does for awainers. At the time of this writing, it doesn't, but who knows what the future could hold?

Finally, there are temporary stack variables. These are introduced when an await expression is used as part of a bigger expression and some intermediate values need to be remembered. Our simple example in listing 6.1 doesn't need any, which is why listing 6.2 shows only four fields: the state, builder, awainer, and parameter. As an example of this, consider the following method:

```
public async Task TemporaryStackDemoAsync()
{
    Task<int> task = Task.FromResult(10);
    DateTime now = DateTime.UtcNow;
    int result = now.Second + now.Hours * await task;
}
```

The C# rules for operand evaluation don't change just because you're within an async method. The properties `now.Second` and `now.Hours` both have to be evaluated before the task is awaited, and their results have to be remembered in order to perform

the arithmetic later, after the state machine resumes when the task completes. That means it needs to use fields.

NOTE In this case, you know that `Task.FromResult` always returns a completed task. But the compiler doesn't know that, and it has to generate the state machine in a way that would let it pause and resume if the task weren't complete.

You can think of it as if the compiler rewrites the code to introduce extra local variables:

```
public async Task TemporaryStackDemoAsync()
{
    Task<int> task = Task.FromResult(10);
    DateTime now = DateTime.UtcNow;
    int tmp1 = now.Second;
    int tmp2 = now.Hours;
    int result = tmp1 + tmp2 * await task;
}
```

Then the local variables are converted into fields. Unlike real local variables, the compiler does reuse temporary stack variables of the same type and generates only as many fields as it needs to.

That explains all the fields in the state machine. Next, you need to look at the `MoveNext()` method—but only conceptually, to start with.

6.1.3 *The MoveNext() method (high level)*

I'm not going to show you the decompiled code for listing 6.1's `MoveNext()` method yet, because it's long and scary.¹ After you know what the flow looks like, it's more manageable, so I'll describe it in the abstract here.

Each time `MoveNext()` is called, the state machine takes another step. Each time it reaches an `await` expression, it'll continue if the value being awaited has already completed and pause otherwise. `MoveNext()` returns if any of the following occurs:

- The state machine needs to pause to await an incomplete value.
- Execution reaches the end of the method or a return statement.
- An exception is thrown but not caught in the `async` method.

Note that in the final case, the `MoveNext()` method doesn't end up throwing an exception. Instead, the task associated with the `async` call becomes faulted. (If that surprises you, see section 5.6.5 for a reminder of the behavior of `async` methods with respect to exceptions.)

Figure 6.2 shows a general flowchart of an `async` method that focuses on the `MoveNext()` method. I haven't included exception handling in the figure, as flowcharts don't have a way of representing `try/catch` blocks. You'll see how that's

¹ If *A Few Good Men* had been about `async`, the line would have been, "You want the `MoveNext`? You can't handle the `MoveNext`!"

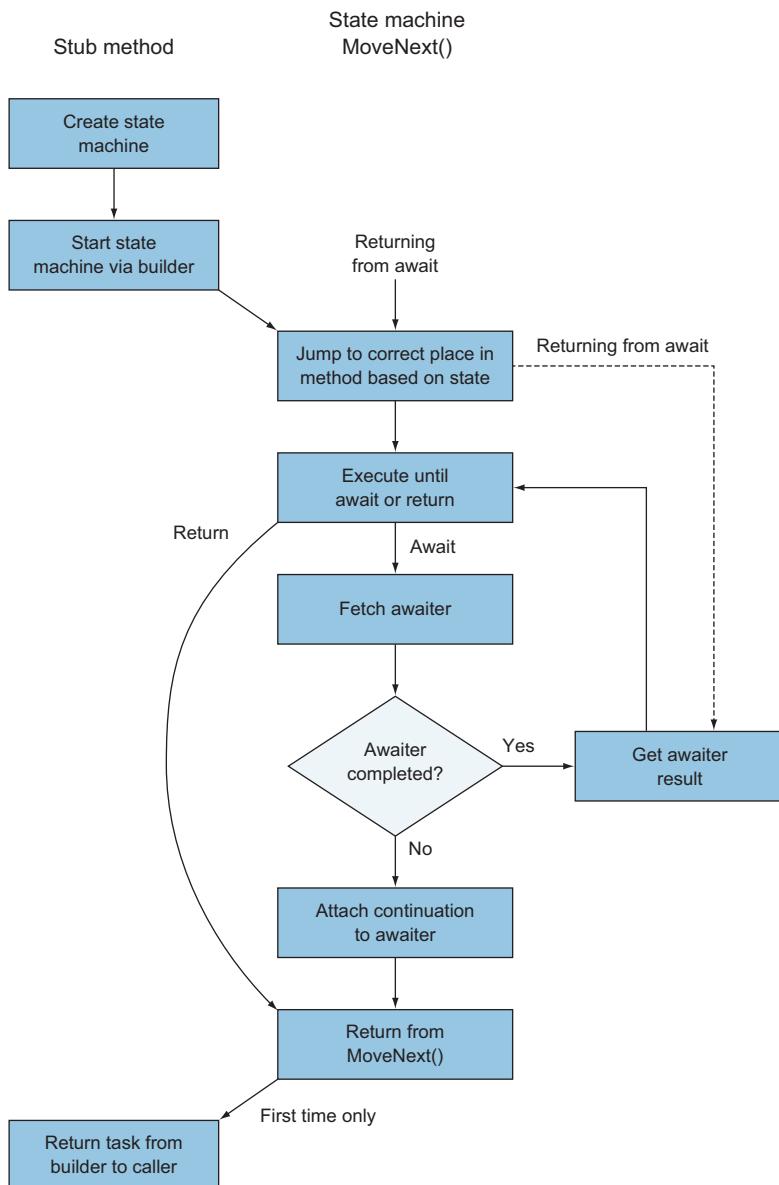


Figure 6.2 Flowchart of an async method

managed when you eventually look at the code. Likewise, I haven't shown where `SetStateMachine` is called, as the flowchart is complicated enough as it is.

One final point about the `MoveNext()` method: its return type is `void`, not a task type. Only the stub method needs to return the task, which it gets from the state machine's builder after the builder's `Start()` method has called `MoveNext()` to take

the first step. All the other calls to `MoveNext()` are part of the infrastructure for resuming the state machine from a paused state, and those don't need the associated task. You'll see what all of this looks like in code in section 6.2 (not long to go now), but first, a brief word on `SetStateMachine`.

6.1.4 ***The SetStateMachine method and the state machine boxing dance***

I've already shown the implementation of `SetStateMachine`. It's simple:

```
void IAsyncStateMachine.SetStateMachine(
    IAsyncStateMachine stateMachine)
{
    this.builder.SetStateMachine(stateMachine);
}
```

The implementation in release builds always looks like this. (In debug builds, where the state machine is a class, the implementation is empty.) The purpose of the method is easy to explain at a high level, but the details are fiddly. When a state machine takes its first step, it's on the stack as a local variable of the stub method. If it pauses, it has to box itself (onto the heap) so that all that information is still in place when it resumes. After it's been boxed, `SetStateMachine` is called on the boxed value using the boxed value as the argument. In other words, somewhere deep in the heart of the infrastructure, there's code that looks a bit like this:

```
void BoxAndRemember<TStateMachine>(ref TStateMachine stateMachine)
    where TStateMachine : IStateMachine
{
    IStateMachine boxed = stateMachine;
    boxed.SetStateMachine(boxed);
}
```

It's not quite as simple as that, but that conveys the essence of what's going on. The implementation of `SetStateMachine` then makes sure that the `AsyncTaskMethodBuilder` has a reference to the single boxed version of the state machine that it's a part of. The method has to be called on the boxed value; it can be called only after boxing, because that's when you have the reference to the boxed value, and if you called it on the unboxed value after boxing, that wouldn't affect the boxed value. (Remember, `AsyncTaskMethodBuilder` is itself a value type.) This intricate dance ensures that when a continuation delegate is passed to the awainer, that continuation will call `MoveNext()` on the same boxed instance.

The result is that the state machine isn't boxed at all if it doesn't need to be and is boxed exactly once if necessary. After it's boxed, everything happens on the boxed version. It's a lot of complicated code in the name of efficiency.

I find this little dance one of the most intriguing and bizarre bits of the whole async machinery. It sounds like it's utterly pointless, but it's necessary because of the way boxing works, and boxing is necessary to preserve information while the state machine is paused.

It's absolutely fine not to fully understand this code. If you ever find yourself debugging async code at a low level, you can come back to this section. For all other intents and purposes, this code is more of a novelty than anything else.

That's what the state machine consists of. Most of the rest of the chapter is devoted to the `MoveNext()` method and how it operates in various situations. We'll start with the simple case and work up from there.

6.2 A simple `MoveNext()` implementation

We're going to start with the simple async method that you saw in listing 6.1. It's simple not because it's short (although that helps) but because it doesn't contain any loops, `try` statements, or `using` statements. It has simple control flow, which leads to a relatively simple state machine. Let's get cracking.

6.2.1 A full concrete example

I'm going to show you the full method to start with. Don't expect this to all make sense yet, but do spend a few minutes looking through it. With this concrete example in hand, the more general structure is easier to understand, because you can always look back to see how each part of that structure is present in this example. At the risk of boring you, here's listing 6.1 yet again as a reminder of the compiler's input:

```
static async Task PrintAndWait(TimeSpan delay)
{
    Console.WriteLine("Before first delay");
    await Task.Delay(delay);
    Console.WriteLine("Between delays");
    await Task.Delay(delay);
    Console.WriteLine("After second delay");
}
```

The following listing is a version of the decompiled code that has been slightly rewritten for readability. (Yes, this is the easy-to-read version.)

Listing 6.3 The decompiled `MoveNext()` method from listing 6.1

```
void IAsyncStateMachine.MoveNext()
{
    int num = this.state;
    try
    {
        TaskAwaiter awaite1;
        switch (num)
        {
            default:
                goto MethodStart;
            case 0:
                goto FirstAwaitContinuation;
            case 1:
                goto SecondAwaitContinuation;
        }
    }
}
```

```

MethodStart:
    Console.WriteLine("Before first delay");
    awaier1 = Task.Delay(this.delay).GetAwaiter();
    if (awaier1.IsCompleted)
    {
        goto GetFirstAwaitResult;
    }
    this.state = num = 0;
    this.awaier = awaier1;
    this.builder.AwaitUnsafeOnCompleted(ref awaier1, ref this);
    return;
FirstAwaitContinuation:
    awaier1 = this.awaier;
    this.awaier = default(TaskAwaiter);
    this.state = num = -1;
GetFirstAwaitResult:
    awaier1.GetResult();
    Console.WriteLine("Between delays");
    TaskAwaiter awaier2 = Task.Delay(this.delay).GetAwaiter();
    if (awaier2.IsCompleted)
    {
        goto GetSecondAwaitResult;
    }
    this.state = num = 1;
    this.awaier = awaier2;
    this.builder.AwaitUnsafeOnCompleted(ref awaier2, ref this);
    return;
SecondAwaitContinuation:
    awaier2 = this.awaier;
    this.awaier = default(TaskAwaiter);
    this.state = num = -1;
GetSecondAwaitResult:
    awaier2.GetResult();
    Console.WriteLine("After second delay");
}
catch (Exception exception)
{
    this.state = -2;
    this.builder.SetException(exception);
    return;
}
this.state = -2;
this.builder.SetResult();
}

```

That's a lot of code, and you may notice that it has a lot of `goto` statements and code labels, which you hardly ever see in handwritten C#. At the moment, I expect it to be somewhat impenetrable, but I wanted to show you a concrete example to start with, so you can refer to it anytime it's useful to you. I'm going to break this down further into general structure and then the specifics of await expressions. By the end of this section, listing 6.3 will probably still look extremely ugly to you, but you'll be in a better position to understand what it's doing and why.

6.2.2 MoveNext() method general structure

We're into the next layer of the async onion. The `MoveNext()` method is at the heart of the async state machine, and its complexity is a reminder of how hard it is to get async code right. The more complex the state machine, the more reason you have to be grateful that it's the C# compiler that has to write the code rather than you.

NOTE It's time to introduce more terminology for the sake of brevity. At each await expression, the value being awaited may already have completed or may still be incomplete. If it has already completed by the time you await it, the state machine keeps executing. I call this the *fast path*. If it hasn't already completed, the state machine schedules a continuation and pauses. I call this the *slow path*.

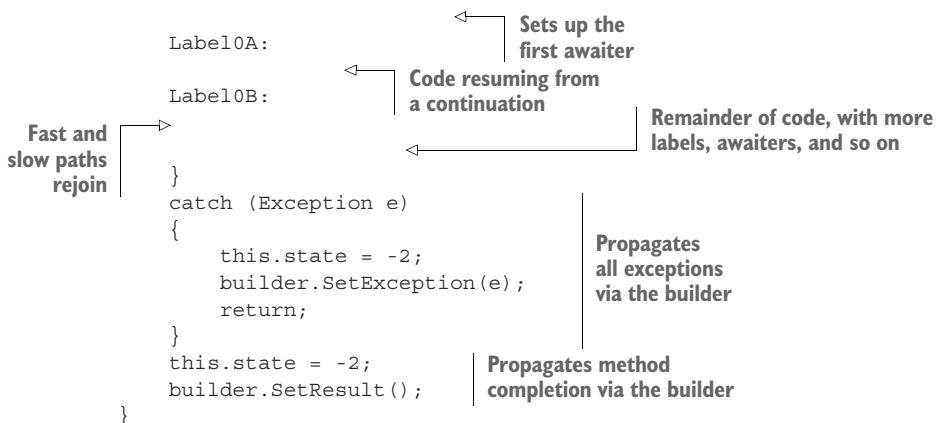
As a reminder, the `MoveNext()` method is invoked once when the async method is first called and then once each time it needs to resume from being paused at an await expression. (If every await expression takes the fast path, `MoveNext()` will be called only once.) The method is responsible for the following:

- Executing from the right place (whether that's the start of the original async code or partway through)
- Preserving state when it needs to pause, both in terms of local variables and location within the code
- Scheduling a continuation when it needs to pause
- Retrieving return values from awaiters
- Propagating exceptions via the builder (rather than letting `MoveNext()` itself fail with an exception)
- Propagating any return value or method completion via the builder

With this in mind, the following listing shows pseudocode for the general structure of a `MoveNext()` method. You'll see in later sections how this can end up being more complicated because of extra control flow, but it's a natural extension.

Listing 6.4 Pseudocode of a MoveNext() method

```
void IAsyncStateMachine.MoveNext()
{
    try
    {
        switch (this.state)
        {
            default: goto MethodStart;
            case 0: goto Label0A;
            case 1: goto Label1A;
            case 2: goto Label2A;
        }
    }
    MethodStart:
        As many cases as there
        are await expressions
        Code before the first
        await expression
    }
```



The big `try/catch` block covers all the code from the original `async` method. If anything in there throws an exception, however it's thrown (via awaiting a faulted operation, calling a synchronous method that throws, or simply throwing an exception directly), that exception is caught and then propagated via the builder. Only special exceptions (`ThreadAbortException` and `StackOverflowException`, for example) will ever cause `MoveNext()` to end with an exception.

Within the `try/catch` block, the start of the `MoveNext()` method is always effectively a `switch` statement used to jump to the right piece of code within the method based on the state. If the state is non-negative, that means you're resuming after an `await` expression. Otherwise, it's assumed that you're executing `MoveNext()` for the first time.

What about other states?

In section 6.1, I listed the possible states as not started, executing, paused, and complete (where paused is a separate state per `await` expression). Why doesn't the state machine handle not started, executing, and complete differently?

The answer is that `MoveNext()` should never end up being called in the executing or complete states. You can force it to by writing a broken awainer implementation or by using reflection, but under normal operation, `MoveNext()` is called only to start or resume the state machine. There aren't even distinct state numbers for not started and executing; both use `-1`. There's a state number of `-2` for completed, but the state machine never checks for that value.

One bit of trickiness to be aware of is the difference between a `return` statement in the state machine and a `return` statement in the original `async` code. Within the state machine, `return` is used when the state machine is paused after scheduling a continuation for an awainer. Any `return` statement in the original code ends up dropping to the bottom part of the state machine outside the `try/catch` block, where the method completion is propagated via the builder.

If you compare listings 6.3 and 6.4, hopefully you can see how our concrete example fits into the general pattern. At this point, I've explained almost everything about the code generated by the simple async method you started with. The only bit that's missing is exactly what happens around await expressions.

6.2.3 Zooming into an await expression

Let's think again about what has to happen each time you hit an await expression when executing an async method, assuming you've already evaluated the operand to get something that's awaitable:

- 1 You fetch the awainer from the awaitable by calling `GetAwaiter()`, storing it on the stack.
- 2 You check whether the awainer has already completed. If it has, you can skip straight to fetching the result (step 9). This is the fast path.
- 3 It looks like you're on the slow path. Oh well. Remember where you reached via the state field.
- 4 Remember the awainer in a field.
- 5 Schedule a continuation with the awainer, making sure that when the continuation is executed, you'll be back to the right state (doing the boxing dance, if necessary).
- 6 Return from the `MoveNext()` method either to the original caller, if this is the first time you've paused, or to whatever scheduled the continuation otherwise.
- 7 When the continuation fires, set your state back to *running* (value of -1).
- 8 Copy the awainer out of the field and back onto the stack, clearing the field in order to potentially help the garbage collector. Now you're ready to rejoin the fast path.
- 9 Fetch the result from the awainer, which is on the stack at this point regardless of which path you took. You have to call `GetResult()` even if there isn't a result *value* to let the awainer propagate errors if necessary.
- 10 Continue on your merry way, executing the rest of the original code using the result value if there was one.

With that list in mind, let's review a section of listing 6.3 that corresponds to our first await expression.

Listing 6.5 A section of listing 6.3 corresponding to a single await

```
awaiter1 = Task.Delay(this.delay).GetAwaiter();
if (awaiter1.IsCompleted)
{
    goto GetFirstAwaitResult;
}
this.state = num = 0;
this.awainer = awaiter1;
this.builder.AwaitUnsafeOnCompleted(ref awaiter1, ref this);
return;
```

```

FirstAwaitContinuation:
    awaier1 = this.awaier;
    this.awaier = default(TaskAwaier);
    this.state = num = -1;
GetFirstAwaitResult:
    awaier1.GetResult();

```

Unsurprisingly, the code follows the set of steps precisely.² The two labels represent the two places you have to jump to, depending on the path:

- In the fast path, you jump over the slow-path code.
- In the slow path, you jump back into the middle of the code when the continuation is called. (Remember, that's what the `switch` statement at the start of the method is for.)

The call to `builder.AwaitUnsafeOnCompleted(ref awaier1, ref this)` is the part that does the boxing dance with a call back into `SetStateMachine` (if necessary; it happens only once per state machine) and schedules the continuation. In some cases, you'll see a call to `AwaitOnCompleted` instead of `AwaitUnsafeOnCompleted`. These differ only in terms of how the execution context is handled. You'll look at this in more detail in section 6.5.

One aspect that may seem slightly unclear is the use of the `num` local variable. It's always assigned a value at the same time as the `state` field but is always read instead of the field. (Its initial value is copied out of the field, but that's the only time the field is read.) I believe this is purely for optimization. Whenever you read `num`, it's fine to think of it as `this.state` instead.

Looking at listing 6.5, that's 16 lines of code for what was originally just the following:

```
await Task.Delay(delay);
```

The good news is that you almost never need to see all that code unless you're going through this kind of exercise. There's a small amount of bad news in that the code inflation means that even small async methods—even those using `ValueTask<TResult>`—can't be sensibly inlined by the JIT compiler. In most cases, that's a minuscule price to pay for the benefits afforded by `async/await`, though.

That's the *simple* case with *simple* control flow. With that background, you can explore a couple of more-complex cases.

6.3 How control flow affects `MoveNext()`

The example you've been looking at so far has just been a sequence of method calls with only the `await` operator introducing complexity. Life gets a little harder when you want to write real code with all the normal control-flow statements you're used to.

In this section, I'll show you just two elements of control flow: loops and `try/finally` statements. This isn't intended to be comprehensive, but it should give you

² It's unsurprising in that it would have been pretty odd of me to write that list of steps and then present code that didn't follow the list.

enough of a glimpse at the control-flow gymnastics the compiler has to perform to help you understand other situations if you need to.

6.3.1 Control flow between await expressions is simple

Before we get into the tricky part, I'll give an example of where introducing control flow doesn't add to the generated code complexity any more than it would in the synchronous code. In the following listing, a loop is introduced into our example method, so you print `Between delays` three times instead of once.

Listing 6.6 Introducing a loop between await expressions

```
static async Task PrintAndWaitWithSimpleLoop(TimeSpan delay)
{
    Console.WriteLine("Before first delay");
    await Task.Delay(delay);
    for (int i = 0; i < 3; i++)
    {
        Console.WriteLine("Between delays");
    }
    await Task.Delay(delay);
    Console.WriteLine("After second delay");
}
```

What does this look like when decompiled? Very much like listing 6.2! The only difference is this

```
GetFirstAwaitResult:
awaiter1.GetResult();
Console.WriteLine("Between delays");
TaskAwaiter awaiter2 = Task.Delay(this.delay).GetAwaiter();
```

becomes the following:

```
GetFirstAwaitResult:
awaiter1.GetResult();
for (int i = 0; i < 3; i++)
{
    Console.WriteLine("Between delays");
}
TaskAwaiter awaiter2 = Task.Delay(this.delay).GetAwaiter();
```

The change in the state machine is exactly the same as the change in the original code. There are no extra fields and no complexities in terms of how to continue execution; it's just a loop.

The reason I bring this up is to help you think about why extra complexity is required in our next examples. In listing 6.6, you never need to jump into the loop from outside, and you never need to pause execution and jump out of the loop, thereby pausing the state machine. Those are the situations introduced by await expressions when you await *within* the loop. Let's do that now.

6.3.2 Awaiting within a loop

Our example so far has contained two await expressions. To keep the code somewhat manageable as I introduce other complexities, I'm going to reduce that to one. The following listing shows the async method you're going to decompile in this subsection.

Listing 6.7 Awaiting in a loop

```
static async Task AwaitInLoop(TimeSpan delay)
{
    Console.WriteLine("Before loop");
    for (int i = 0; i < 3; i++)
    {
        Console.WriteLine("Before await in loop");
        await Task.Delay(delay);
        Console.WriteLine("After await in loop");
    }
    Console.WriteLine("After loop delay");
}
```

The `Console.WriteLine` calls are mostly present as signposts within the decompiled code, which makes it easier to map to the original listing.

What does the compiler generate for this? I'm not going to show the complete code, because most of it is similar to what you've seen before. (It's all in the downloadable source, though.) The stub method and state machine are almost exactly as they were for earlier examples but with one additional field in the state machine corresponding to `i`, the loop counter. The interesting part is in `MoveNext()`.

You can represent the code faithfully in C# but not using a loop construct. The problem is that after the state machine returns from pausing at `Task.Delay`, you want to jump into the middle of the original loop. You can't do that with a `goto` statement in C#; the language forbids a `goto` statement specifying a label if the `goto` statement isn't in the scope of that label.

That's okay; you can implement your `for` loop with a lot of `goto` statements without introducing any extra scopes at all. That way, you can jump to the middle of it without a problem. The following listing shows the bulk of the decompiled code for the body of the `MoveNext()` method. I've included only the part within the `try` block, as that's what we're focusing on here. (The rest is simple boilerplate.)

Listing 6.8 Decompiled loop without using any loop constructs

```
switch (num)
{
    default:
        goto MethodStart;
    case 0:
        goto AwaitContinuation;
}
MethodStart:
    Console.WriteLine("Before loop");
```

```

this.i = 0;           ← For loop initializer
Skips straight      goto ForLoopCondition;
to checking          ←
the loop             ForLoopBody:
condition:
Console.WriteLine("Before await in loop");
TaskAwaiter awaiter = Task.Delay(this.delay).GetAwaiter();
if (awaiter.IsCompleted)
{
    goto GetAwaitResult;
}
this.state = num = 0;
this.awaiter = awaiter;
this.builder.AwaitUnsafeOnCompleted(ref awaiter, ref this);
return;
AwaitContinuation:
awaiter = this.awaiter;
this.awaiter = default(TaskAwaiter);
this.state = num = -1;
GetAwaitResult:
awaiter.GetResult();
Console.WriteLine("After await in loop");
this.i++;
ForLoopCondition:
if (this.i < 3)
{
    goto ForLoopBody;
}
Console.WriteLine("After loop delay");

```

Body of the for loop

Target for jump when the state machine resumes

For loop iterator

Checks for loop condition and jumps back to body if it holds

I could've skipped this example entirely, but it brings up a few interesting points. First, the C# compiler doesn't convert an async method into equivalent C# that doesn't use async/await. It only has to generate appropriate IL. In some places, C# has rules that are stricter than those in IL. (The set of valid identifiers is another example of this.)

Second, although decompilers can be useful when looking at async code, sometimes they produce invalid C#. When I first decompiled the output of listing 6.7, the output included a while loop containing a label and a goto statement outside that loop trying to jump into it. You can sometimes get valid (but harder-to-read) C# by telling the decompiler not to work as hard to produce idiomatic C#, at which point you'll see an awful lot of goto statements.

Third, in case you weren't already convinced, you don't want to be writing this sort of code by hand. If you had to write C# 4 code for this sort of task, you'd no doubt do it in a very different way, but it would still be significantly uglier than the async method you can use in C# 5.

You've seen how awaiting within a loop might cause humans some stress, but it doesn't cause the compiler to break a sweat. For our final control-flow example, you'll give it some harder work to do: a try/finally block.

6.3.3 Awaiting within a try/finally block

Just to remind you, it's always been valid to use await in a try block, but in C# 5, it was invalid to use it in a catch or finally block. That restriction was lifted in C# 6, although I'm not going to show any code that takes advantage of it.

NOTE There are simply too many possibilities to go through here. The aim of this chapter is to give you insight into the kind of thing the C# compiler does with `async/await` rather than provide an exhaustive list of translations.

In this section, I'm only going to show you an example of awaiting within a `try` block that has just a `finally` block. That's probably the most common kind of `try` block, because it's the one that `using` statements are equivalent to. The following listing shows the `async` method you're going to decompile. Again, all the console output is present only to make it simpler to understand the state machine.

Listing 6.9 Awaiting within a try block

```
static async Task AwaitInTryFinally(TimeSpan delay)
{
    Console.WriteLine("Before try block");
    await Task.Delay(delay);
    try
    {
        Console.WriteLine("Before await");
        await Task.Delay(delay);
        Console.WriteLine("After await");
    }
    finally
    {
        Console.WriteLine("In finally block");
    }
    Console.WriteLine("After finally block");
}
```

You might imagine that the decompiled code would look something like this:

```
switch (num)
{
    default:
        goto MethodStart;
    case 0:
        goto AwaitContinuation;
}
MethodStart:
...
try
{
    ...
AwaitContinuation:
    ...
GetAwaitResult:
    ...
}
    finally
{
    ...
}
...
```

Here, each ellipsis (...) represents more code. There's a problem with that approach, though: even in IL, you're not allowed to jump from outside a `try` block to inside it. It's a little bit like the problem you saw in the previous section with loops, but this time instead of a C# rule, it's an IL rule.

To achieve this, the C# compiler uses a technique I like to think of as a *trampoline*. (This isn't official terminology, although the term is used elsewhere for similar purposes.) It jumps to just before the `try` block, and then the first thing inside the `try` block is a piece of code that jumps to the right place within the block.

In addition to the trampoline, the `finally` block needs to be handled with care, too. There are three situations in which you'll execute the `finally` block of the generated code:

- You reach the end of the `try` block.
 - The `try` block throws an exception.
 - You need to pause within the `try` block because of an await expression.

(If the `async` method contained a `return` statement, that would be another option.) If the `finally` block is executing because you're pausing the state machine and returning to the caller, the code in the original `async` method's `finally` block shouldn't execute. After all, you're logically paused inside the `try` block and will be resuming there when the delay completes. Fortunately, this is easy to detect: the `num` local variable (which always has the same as the `state` field) is negative if the state machine is still executing or finished and non-negative if you're pausing.

All of this together leads to the following listing, which again is the code within the outer `try` block of `MoveNext()`. Although there's still a lot of code, most of it is similar to what you've seen before. I've highlighted the `try/finally`-specific aspects in bold.

Listing 6.10 Decompile await within try/finally

```
switch (num)
{
    default:
        goto MethodStart;
    case 0:
        goto AwaitContinuationTrampoline; ←
}
MethodStart:
Console.WriteLine("Before try");
AwaitContinuationTrampoline:
try
{
    switch (num)
    {
        default:
            goto TryBlockStart;
        case 0:
            goto AwaitContinuation;
    }
}
TryBlockStart:
```

Jumps to just before the trampoline, so it can bounce execution to the right place

Trampoline within the try block

```

Console.WriteLine("Before await");
TaskAwaiter awaier = Task.Delay(this.delay).GetAwaiter();
if (awaier.IsCompleted)
{
    goto GetAwaitResult;
}
this.state = num = 0;
this.awaier = awaier;
this.builder.AwaitUnsafeOnCompleted(ref awaier, ref this);
return;
AwaitContinuation:
awaier = this.awaier;
this.awaier = default(TaskAwaiter);
this.state = num = -1;
GetAwaitResult:
awaier.GetResult();
Console.WriteLine("After await");
}
finally
{
    if (num < 0)
    {
        Console.WriteLine("In finally block");
    }
}
Console.WriteLine("After finally block");

```

Real continuation target

Effectively ignores finally block if you're pausing

That's the final decompilation in the chapter, I promise. I wanted to get to that level of complexity to help you navigate the generated code if you ever need to. That's not to say you won't need to keep your wits about you when looking through it, particularly bearing in mind the many transformations the compiler can perform to make the code simpler than what I've shown. As I said earlier, where I've always used a switch statement for "jump to X" pieces of code, the compiler can sometimes use simpler branching code. Consistency in multiple situations is important when reading source code, but that doesn't matter to the compiler.

One of the aspects I've skimmed over so far is why awaiters have to implement `INotifyCompletion` but can also implement `ICriticalNotifyCompletion`, and the effect that has on the generated code. Let's take a closer look now.

6.4 Execution contexts and flow

In section 5.2.2, I described synchronization contexts, which are used to govern the thread that code executes on. This is just one of many contexts in .NET, although it's probably the best known. Context provides an ambient way of maintaining information transparently. For example, `SecurityContext` keeps track of the current security principal and code access security. You don't need to pass all that information around explicitly; it just follows your code, doing the right thing in almost all cases. A single class is used to manage all the other contexts: `ExecutionContext`.

Deep and scary stuff

I almost didn't include this section. It's at the very limits of my knowledge about async. If you ever need to know the intimate details, you'll want to know far more about the topic than I've included here.

I've covered this at all only because otherwise there'd be no explanation whatsoever for having both `AwaitOnCompleted` and `AwaitUnsafeOnCompleted` in the builder or why awaiters usually implement `ICriticalNotifyCompletion`.

As a reminder, `Task` and `Task<T>` manage the synchronization context for any tasks being awaited. If you're on a UI thread and you await a task, the continuation of your async method will be executed on the UI thread, too. You can opt out of that by using `Task.ConfigureAwait`. You need that in order to explicitly say "I know I don't need the rest of my method to execute in the same synchronization context." Execution contexts aren't like that; you pretty much always want the same execution context when your async method continues, even if it's on a different thread.

This preservation of the execution context is called *flow*. An execution context is said to flow across await expressions, meaning that all your code operates in the same execution context. What makes sure that happens? Well, `AsyncTaskMethodBuilder` always does, and `TaskAwaiter` sometimes does. This is where things get tricky.

The `INotifyCompletion.OnCompleted` method is just a normal method; anyone can call it. By contrast, `ICriticalNotifyCompletion.UnsafeOnCompleted` is marked with `[SecurityCritical]`. It can be called only by trusted code, such as the framework's `AsyncTaskMethodBuilder` class.

If you ever write your own awaiter class and you care about running code correctly and safely in partially trusted environments, you should ensure that your `INotifyCompletion.OnCompleted` code flows the execution context (via `ExecutionContext.Capture` and `ExecutionContext.Run`). You can also implement `ICriticalNotifyCompletion` and not flow the execution context in that case, trusting that the async infrastructure will already have done so. Effectively, this is an optimization for the common case in which awaiters are used only by the async infrastructure. There's no point in capturing and restoring the execution context twice in cases where you can safely do it only once.

When compiling an async method, the compiler will create a call to either `builder.AwaitOnCompleted` or `builder.AwaitUnsafeOnCompleted` at each await expression, depending on whether the awaiter implements `ICriticalNotifyCompletion`. Those builder methods are generic and have constraints to ensure that the awaiters that are passed into them implement the appropriate interface.

If you ever implement your own custom task type (and again, that's extremely unlikely for anything other than educational purposes), you should follow the same pattern as `AsyncTaskMethodBuilder`: capture the execution context in both

`AwaitOnCompleted` and `AwaitUnsafeOnCompleted`, so it's safe to call `ICriticalNotifyCompletion.UnsafeOnCompleted` when you're asked to. Speaking of custom tasks, let's review the requirements for a custom task builder now that you've seen how the compiler uses `AsyncTaskMethodBuilder`.

6.5 Custom task types revisited

Listing 6.11 shows a repeat of the builder part of listing 5.10, where you first looked at custom task types. The set of methods may feel a lot more familiar now after you've looked at so many decompiled state machines. You can use this section as a reminder of how the methods on `AsyncTaskMethodBuilder` are called, as the compiler treats all builders the same way.

Listing 6.11 A sample custom task builder

```
public class CustomTaskBuilder<T>
{
    public static CustomTaskBuilder<T> Create();
    public void Start<TStateMachine>(ref TStateMachine stateMachine)
        where TStateMachine : IAsyncStateMachine;
    public CustomTask<T> Task { get; }

    public void AwaitOnCompleted<TAwaiter, TStateMachine>
        (ref TAwaiter awaiter, ref TStateMachine stateMachine)
        where TAwaiter : INotifyCompletion
        where TStateMachine : IAsyncStateMachine;
    public void AwaitUnsafeOnCompleted<TAwaiter, TStateMachine>
        (ref TAwaiter awaiter, ref TStateMachine stateMachine)
        where TAwaiter : INotifyCompletion
        where TStateMachine : IAsyncStateMachine;
    public void SetStateMachine(IAsyncStateMachine stateMachine);

    public void SetException(Exception exception);
    public void SetResult(T result);
}
```

I've grouped the methods in the normal chronological order in which they're called.

The stub method calls `Create` to create a builder instance as part of the newly created state machine. It then calls `Start` to make the state machine take the first step and returns the result of the `Task` property.

Within the state machine, each await expression will generate a call to `AwaitOnCompleted` or `AwaitUnsafeOnCompleted` as discussed in the previous section. Assuming a task-like design, the first such call will end up calling `IAsyncStateMachine.SetStateMachine`, which will in turn call the builder's `SetStateMachine` so that any boxing is resolved in a consistent way. See section 6.1.4 for a reminder of the details.

Finally, a state machine indicates that the async operation has completed by calling either `SetException` or `SetResult` on the builder. That final state should be propagated to the custom task that was originally returned by the stub method.

This chapter is by far the deepest dive in this book. Nowhere else do I look at the code generated by the C# compiler in such detail. To many developers, everything in this chapter would be superfluous; you don't really need it to write correct async code in C#. But for curious developers, I hope it's been enlightening. You may never need to decompile generated code, but having some idea of what's going on under the hood can be useful. And if you ever do need to look at what's going on in detail, I hope this chapter will help you make sense of what you see.

I've taken two chapters to cover the one major feature of C# 5. In the next short chapter, I'll cover the remaining two features. After the details of async, they come as a bit of light relief.

Summary

- Async methods are converted into stub methods and state machines by using builders as async infrastructure.
- The state machine keeps track of the builder, method parameters, local variables, awaiters, and where to resume in a continuation.
- The compiler creates code to get back into the middle of a method when it resumes.
- The `INotifyCompletion` and `ICriticalNotifyCompletion` interfaces help control execution context flow.
- The methods of custom task builders are called by the C# compiler.



C# 5 bonus features

This chapter covers

- Changes to variable capture in `foreach` loops
- Caller information attributes

If C# had been designed with book authors in mind, this chapter wouldn't exist, or it'd be a more standard length. I could claim that I wanted to include a very short chapter as a sort of palette cleanser after the dish of asynchrony served by C# 5 and before the sweetness of C# 6, but the reality is that two more changes in C# 5 that need to be covered wouldn't fit into the async chapters. The first of these isn't so much a feature as a correction to an earlier mistake in the language design.

7.1 **Capturing variables in `foreach` loops**

Before C# 5, `foreach` loops were described in the language specification as if each loop declared a single *iteration variable*, which was read-only within the original code but received a different value for each iteration of the loop. For example, in C# 3 a `foreach` loop over a `List<string>` like this

```
foreach (string name in names)
{
```

```

        Console.WriteLine(name);
    }
}

```

would be broadly equivalent to this:

```

string name;
using (var iterator = names.GetEnumerator())
{
    while (iterator.MoveNext())
    {
        name = iterator.Current;
        Console.WriteLine(name);
    }
}

```

NOTE The specification has a lot of other details around possible conversions of both the collection and the elements, but they're not relevant to this change. Additionally, the scope of the iteration variable is only the scope of the loop; you can imagine adding an extra pair of curly braces around the whole code.

In C# 1, this was fine, but it started causing problems way back in C# 2 when anonymous methods were introduced. That was the first time that a variable could be *captured*, changing its lifetime significantly. A variable is captured when it's used in an anonymous function, and the compiler has to do work behind the scenes to make its use feel natural. Although anonymous methods in C# 2 were useful, my impression is that it was C# 3, with its lambda expressions and LINQ, that really encouraged developers to use delegates more widely.

What's the problem with our earlier expansion of the `foreach` loop using just a single iteration variable? If that iteration variable is captured in an anonymous function for a delegate, then whenever the delegate is invoked, the delegate will use the current value of that single variable. The following listing shows a concrete example.

Listing 7.1 Capturing the iteration variable in a foreach loop

```

List<string> names = new List<string> { "x", "y", "z" };
var actions = new List<Action>();
foreach (string name in names)
{
    actions.Add(() => Console.WriteLine(name));
}
foreach (Action action in actions)
{
    action();
}

```

What would you have expected that to print out if I hadn't been drawing your attention to the problem? Most developers would expect it to print x, then y, then z. That's

the useful behavior. In reality, with a C# compiler before version 5, it would've printed z three times, which is really not helpful.

As of C# 5, the specification for the `foreach` loop has been changed so that a new variable is introduced in each iteration of the loop. The exact same code in C# 5 and later produces the expected result of x, y, z.

Note that this change affects only `foreach` loops. If you were to use a regular `for` loop instead, you'd still capture only a single variable. The following listing is the same as listing 7.1, other than the changes shown in bold.

Listing 7.2 Capturing the iteration variable in a `for` loop

```
List<string> names = new List<string> { "x", "y", "z" };
var actions = new List<Action>();
for (int i = 0; i < names.Count; i++)           ← Iterates over the
{                                                 list of names
    actions.Add(() => Console.WriteLine(names[i]));
}
foreach (Action action in actions)               ← Creates a delegate that
{                                                 captures names and i
    action();                                     ← Executes all
}                                                 the delegates
```

This doesn't print the last name three times; it fails with an `ArgumentOutOfRangeException`, because by the time you start executing the delegates, the value of `i` is 3.

This isn't an oversight on the part of the C# design team. It's just that when a `for` loop initializer declares a local variable, it does so once for the whole duration of the loop. The syntax of the loop makes that model easy to see, whereas the syntax of `foreach` encourages a mental model of one variable per iteration. On to our final feature of C# 5: caller information attributes.

7.2 Caller information attributes

Some features are general, such as lambda expressions, implicitly typed local variables, generics, and the like. Others are more specific: LINQ is meant to be about querying data of some form or other, even though it's aimed to generalize over many data sources. The final C# 5 feature is extremely targeted: there are two significant use cases (one obvious, one slightly less so), and I don't expect it to be used much outside those situations.

7.2.1 Basic behavior

.NET 4.5 introduced three new attributes:

- `CallerFilePathAttribute`
- `CallerLineNumberAttribute`
- `CallerMemberNameAttribute`

These are all in the `System.Runtime.CompilerServices` namespace. Just as with other attributes, when you apply any of these, you can omit the `Attribute` suffix. Because that's the most common way of using attributes, I'll abbreviate the names appropriately for the rest of the book.

All three attributes can be applied only to parameters, and they're useful only when they're applied to optional parameters with appropriate types. The idea is simple: if the call site doesn't provide the argument, the compiler will use the current file, line number, or member name to fill in the argument instead of taking the normal default value. If the caller does supply an argument, the compiler will leave it alone.

NOTE The parameter types are almost always `int` or `string` in normal usage. They can be other types where appropriate conversions are available. See the specification for details if you're interested, but I'd be surprised if you ever needed to know.

The following listing is an example of all three attributes and a mixture of compiler-specified and user-specified values.

Listing 7.3 Basic demonstration of caller member attributes

```
static void ShowInfo(
    [CallerFilePath] string file = null,
    [CallerLineNumber] int line = 0,
    [CallerMemberName] string member = null)
{
    Console.WriteLine("{0}:{1} - {2}", file, line, member);
}

static void Main()
{
    ShowInfo();                                ← Compiler provides all three
    ShowInfo("LiesAndDamnedLies.java", -10);   ← arguments from context
}
```

Compiler provides only the member name from context

The output of listing 7.3 on my machine is as follows:

```
C:\Users\jon\Projects\CSharpInDepth\Chapter07\CallerInfoDemo.cs:20 - Main
LiesAndDamnedLies.java:-10 - Main
```

You wouldn't usually give a fake value for any of these arguments, but it's useful to be able to pass the value explicitly, particularly if you want to log the current method's caller using the same attributes.

The member name works for all members normally in the obvious way. The default values for the attributes are usually irrelevant, but we'll come back to some interesting corner cases in section 7.2.4. First, we'll look at the two common use cases I mentioned earlier. The most universal of these is logging.

7.2.2 Logging

The most obvious case in which caller information is useful is when writing to a log file. Previously when logging, you'd usually construct a stack trace (using `System.Diagnostics.StackTrace`, for example) to find out where the log call came from. This is typically hidden from view in logging frameworks, but it's still there—and ugly. It's potentially an issue in terms of performance, and it's brittle in the face of JIT compiler inlining.

It's easy to see how a logging framework can use the new feature to allow caller-only information to be logged cheaply, even preserving line numbers and member names in the face of a build that had debug information stripped and even after obfuscation. This doesn't help when you want to log a full stack trace, of course, but it doesn't take away your ability to do that, either.

Based on a quick sampling performed at the end of 2017, it appears that this functionality hasn't been used particularly widely yet.¹ In particular, I see no sign of it being used in the `ILogger` interface commonly used in ASP.NET Core. But it'd be entirely reasonable to write your own extension methods for `ILogger` that use these attributes and create an appropriate state object to be logged.

It's not particularly uncommon for projects to include their own primitive logging frameworks, which could also be amenable to the use of these attributes. A project-specific logging framework is less likely to need to worry about targeting frameworks that don't include the attributes, too.

NOTE The lack of an efficient system-level logging framework is a thorny issue. This is particularly true for class library developers who wish to provide logging facilities but don't want to add third-party dependencies and don't know which logging frameworks their users will be targeting.

Whereas the logging use case needs specific thought on the part of frameworks, our second use case is a lot simpler to integrate.

7.2.3 Simplifying `INotifyPropertyChanged` implementations

The less obvious use of just one of these attributes, `[CallerMemberName]`, may be obvious to you if you happen to implement `INotifyPropertyChanged` frequently. If you're not familiar with the `INotifyPropertyChanged` interface, it's commonly used for thick client applications (as opposed to web applications) to allow a user interface to respond to a change in model or view model. It's in the `System.ComponentModel` namespace, so it's not tied to any particular UI technology. It's used in Windows Forms, WPF, and Xamarin Forms, for example. The interface is simple; it's a single event of type `PropertyChangedEventHandler`. This is a delegate type with the following signature:

¹ NLog was the only logging framework I found with direct support and then only conditionally based on the target framework.

```
public delegate void PropertyChangedEventHandler(
    Object sender, PropertyChangedEventArgs e)
```

PropertyChangedEventArgs, in turn, has a single constructor:

```
public PropertyChangedEventArgs(string propertyName)
```

A typical implementation of `INotifyPropertyChanged` before C# 5 might look something like the following listing.

Listing 7.4 Implementing `INotifyPropertyChanged` the old way

```
class OldPropertyNotifier : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;
    private int firstValue;
    public int FirstValue
    {
        get { return firstValue; }
        set
        {
            if (value != firstValue)
            {
                firstValue = value;
                NotifyPropertyChanged("FirstValue");
            }
        }
    }
    // (Other properties with the same pattern)

    private void NotifyPropertyChanged(string propertyName)
    {
        PropertyChangedEventHandler handler = PropertyChanged;
        if (handler != null)
        {
            handler(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

The purpose of the helper method is to avoid having to put the nullity check in each property. You could easily make it an extension method to avoid repeating it on each implementation.

This isn't just long-winded (which hasn't changed); it's also brittle. The problem is that the name of the property (`FirstValue`) is specified as a string literal, and if you refactor the property name to something else, you could easily forget to change the string literal. If you're lucky, your tools and tests will help you spot the mistake, but it's still ugly. You'll see in chapter 9 that the `nameof` operator introduced in C# 6 would make this code more refactoring friendly, but it'd still be prone to copy-and-paste errors.

With caller info attributes, the majority of the code stays the same, but you can make the compiler fill in the property name by using `CallerMemberName` in the helper method, as shown in the following listing.

Listing 7.5 Using caller information to implement `INotifyPropertyChanged`

```
if (value != firstValue)
{
    firstValue = value;
    NotifyPropertyChanged();
}

void NotifyPropertyChanged([CallerMemberName] string propertyName = null)
{
}
```

I've shown only the sections of the code that have changed; it's that simple. Now when you change the name of the property, the compiler will use the new name instead. It's not an earth-shattering improvement, but it's nicer nonetheless.

Unlike logging, this pattern has been embraced by model-view-viewmodel (MVVM) frameworks that provide base classes for view models and models. For example, in Xamarin Forms, the `BindableObject` class has an `OnPropertyChanged` method using `CallerMemberName`. Similarly, the Caliburn Micro MVVM framework has a `PropertyChangedBase` class with a `NotifyOfPropertyChange` method. That's all you're likely to need to know about caller information attributes, but a few interesting oddities exist, particularly with the caller member name.

7.2.4 Corner cases of caller information attributes

In almost all cases, it's obvious which value the compiler should provide for caller information attributes. It's interesting to look at places where it's not obvious, though. I should emphasize that this is mostly a matter of curiosity and a spotlight on language design choices rather than on issues that will affect regular development. First, a little restriction.

DYNAMICALLY INVOKED MEMBERS

In many ways, the infrastructure around dynamic typing tries hard to apply the same rules at execution time as the regular compiler would at compile time. But caller information isn't preserved for this purpose. If the member being invoked includes an optional parameter with a caller information attribute but the invocation doesn't include a corresponding argument, the default value specified in the parameter is used as if the attribute weren't present.

Aside from anything else, the compiler would have to embed all the line-number information for every dynamically invoked member just in case it was required, thereby increasing the resulting assembly size for no benefit in 99.9% of cases. Then there's the

extra analysis required at execution time to check whether the caller information was required, which would potentially disrupt caching, too. I suspect that if the C# design team had considered this to be a common and important scenario, they'd have found a way to make it work, but I also think it's entirely reasonable that they decided there were more valuable features to spend their time on. You just need to be aware of the behavior and accept it, basically. Workarounds exist in some cases, though.

If you're passing a method argument that happens to be dynamic but you don't need it to be, you can cast to the appropriate type instead. At that point, the method invocation will be a regular one without any dynamic typing involved.² If you really need the dynamic behavior but you know that the member you're invoking uses caller information attributes, you can explicitly call a helper method that uses the caller information attribute to return the value. It's a little ugly, but this is a corner case anyway. The following listing shows the problem and both workarounds.

Listing 7.6 Caller information attributes and dynamic typing

```
static void ShowLine(string message,
    [CallerLineNumber] int line = 0)
{
    Console.WriteLine("{0}: {1}", line, message);
}

static int GetLineNumber(
    [CallerLineNumber] int line = 0)
{
    return line;
}

static void Main()
{
    dynamic message = "Some message";
    ShowLine(message);           ← Simple dynamic call;
                                ← line will be reported as 0.
    ShowLine((string) message); ←
    ShowLine(message, GetLineNumber()); ←
}                           ← Workaround I: cast the value
                            ← to remove dynamic typing.

Workaround 2: explicitly provide the
line number using a helper method.
```

Listing 7.6 prints a line number of 0 for the first call but the correct line number for both workarounds. It's a trade-off between having simple code and retaining more information. Neither of these workarounds is appropriate when you need to use dynamic overload resolution, and some overloads need caller information and some don't, of course. As limitations go, that's pretty reasonable in my view. Next, let's think about unusual names.

² The call will have the additional benefits of compile-time checking that the member exists and improved execution-time efficiency, too.

NON-OBVIOUS MEMBER NAMES

When the caller member name is provided by the compiler and that caller is a method, the name is obvious: it's the name of the method. Not everything is a method, though. Here are some cases to consider:

- Calls from an instance constructor
- Calls from a static constructor
- Calls from a finalizer
- Calls from an operator
- Calls as part of a field, event, or property initializer³
- Calls from an indexer

The first four of these are specified to be implementation dependent; it's up to the compiler to decide how to treat them. The fifth (initializers) isn't specified at all, and the final one (indexers) is specified to use the name `Item` unless `IndexerName-Attribute` has been applied to the indexer.

The Roslyn compiler uses the names that are present in the IL for the first four: `.ctor`, `.cctor`, `Finalize`, and operator names such as `op>Addition`. For initializers, it uses the name of the field, event, or property being initialized.

The downloadable code contains a complete example showing all of these; I haven't included the code here, as the results are more interesting than the code itself. All of the names are the most obvious ones to pick, and I'd be surprised to see a different compiler pick a different option. I have found a difference between compilers for another aspect, however: determining when the compiler should fill in caller information attributes at all.

IMPLICIT CONSTRUCTOR INVOCATIONS

The C# 5 language specification requires that caller information be used only when a function is explicitly invoked in source code, with the exception of query expressions that are deemed to be syntactic expansions. Other C# language constructs that are pattern based don't apply to methods with optional parameters anyway, but constructor initializers definitely do. (Deconstruction is a C# 7 feature described in section 12.2.) The language specification calls out constructors as an example in which caller member information isn't provided by the compiler unless the call is explicit. The following listing shows a single abstract base class with a constructor using caller member information and three derived classes.

Listing 7.7 Caller information in a constructor

```
public abstract class BaseClass
{
    protected BaseClass(
        [CallerFilePath] string file = "Unspecified file",
        [CallerMemberName] string memberName = null,
        [CallerLineNumber] int lineNumber = 0)
    {
        // ...
    }
}
```

Base class constructor
uses caller info attributes.

³ Initializers for automatically implemented properties were introduced in C# 6. See section 8.2.2 for details, but if you take a guess at what this means, you're likely to be right.

```

[CallerLineNumber] int line = -1,
[CallerMemberName] string member = "Unspecified member")
{
    Console.WriteLine("{0}:{1} - {2}", file, line, member);
}
}

public class Derived1 : BaseClass { }           ← Parameterless constructor  
is added implicitly.

public class Derived2 : BaseClass
{
    public Derived2() { }           ← Constructor with  
implicit call to base()
}

public class Derived3 : BaseClass
{
    public Derived3() : base() {}   ← Explicit call  
to base
}

```

With Roslyn, only `Derived3` will result in real caller information being shown. Both `Derived1` and `Derived2`, in which the call to the `BaseClass` constructor is implicit, use the default values specified in the parameters rather than providing the filename, line number, and member name.

This is in line with the C# 5 specification, but I'd argue it's a design flaw. I believe most developers would expect the three derived classes to be precisely equivalent. Interestingly, the Mono compiler (`mcs`) currently prints the same output for each of these derived classes. We'll have to wait to see whether the language specification changes, the Mono compiler changes, or the incompatibility continues into the future.

QUERY EXPRESSION INVOCATIONS

As I mentioned before, the language specification calls out query expressions as one place where caller information is provided by the compiler even though the call is implicit. I doubt that this will be used often, but I've provided a complete example in the downloadable source code. It requires more code than would be sensible to include here, but its use looks like the following listing.

Listing 7.8 Caller information in query expressions

```

string[] source =
{
    "the", "quick", "brown", "fox",
    "jumped", "over", "the", "lazy", "dog"
};

var query = from word in source
            where word.Length > 3
            select word.ToUpperInvariant();

Console.WriteLine("Data:");
Console.WriteLine(string.Join(", ", query));
Console.WriteLine("CallerInfo:");
Console.WriteLine(string.Join(
    Environment.NewLine, query.CallerInfo));           ← Logs  
the data

```

Query expression using methods
capturing caller information

Logs
the data

Logs the caller
information of the query

Although it contains a regular query expression, I've introduced new extension methods (in the same namespace as the example, so they're found before the `System.Linq` ones) containing caller information attributes. The output shows that the caller information is captured in the query as well as the data itself:

```
Data:  
QUICK, BROWN, JUMPED, OVER, LAZY  
CallerInfo:  
CallerInfoLinq.cs:91 - Main  
CallerInfoLinq.cs:92 - Main
```

Is this useful? Probably not, to be honest. But it does highlight that when the language designers introduced the feature, they had to carefully consider a lot of situations. It would've been annoying if someone had found a good use for caller information from query expressions, but the specification hadn't made it clear what should happen. We have one final kind of member invocation to consider, which feels to me like it's even more subtle than constructor initializers and query expressions: attribute instantiation.

ATTRIBUTES WITH CALLER INFORMATION ATTRIBUTES

I tend to think about applying attributes as just specifying extra data. It doesn't feel like it's invoking anything, but attributes are code too, and when an attribute object is constructed (usually to be returned from a reflection call), that calls constructors and property setters. What counts as the caller if you create an attribute that uses caller information attributes in its constructor? Let's find out.

First, you need an attribute class. This part is simple and is shown in the following listing.

Listing 7.9 Attribute class that captures caller information

```
[AttributeUsage(AttributeTargets.All)]  
public class MemberDescriptionAttribute : Attribute  
{  
    public MemberDescriptionAttribute(  
        [CallerFilePath] string file = "Unspecified file",  
        [CallerLineNumber] int line = 0,  
        [CallerMemberName] string member = "Unspecified member")  
    {  
        File = file;  
        Line = line;  
        Member = member;  
    }  
  
    public string File { get; }  
    public int Line { get; }  
    public string Member { get; }  
  
    public override string ToString() =>  
        $"{Path.GetFileName(File)}:{Line} - {Member}";  
}
```

For brevity, this class uses a few features from C# 6, but the interesting aspect for now is that the constructor parameters use caller information attributes.

What happens when you apply our new `MemberDescriptionAttribute`? In the next listing, let's apply it to a class and various aspects of a method and then see what you get.

Listing 7.10 Applying the attribute to a class and a method

```
using MDA = MemberDescriptionAttribute;                                ← Helps keep the
[MemberDescription]                                                 reflection code short
class CallerNameInAttribute {                                         | Applies the attribute to a class
{
    [MemberDescription]
    public void Method<[MemberDescription] T>(
        [MemberDescription] int parameter) { }                         | Applies the attribute to
                                                                     a method in various ways
    static void Main()
    {
        var typeInfo = typeof(CallerNameInAttribute).GetTypeInfo();
        var methodInfo = typeInfo.GetDeclaredMethod("Method");
        var pInfo = methodInfo.GetParameters()[0];
        var typeParamInfo =
            methodInfo.GetGenericArguments()[0].GetTypeInfo();
        Console.WriteLine(typeInfo.GetCustomAttribute<MDA>());
        Console.WriteLine(methodInfo.GetCustomAttribute<MDA>());
        Console.WriteLine(pInfo.GetCustomAttribute<MDA>());
        Console.WriteLine(typeParamInfo.GetCustomAttribute<MDA>());
    }
}
```

The `Main` method uses reflection to fetch the attribute from all the places you've applied it. You could apply `MemberDescriptionAttribute` to other places: fields, properties, indexers, and the like. Feel free to experiment with the downloadable code to find out exactly what happens. What I find interesting is that the compiler is perfectly happy to capture the line number and file path in all cases, but it doesn't use the class name as the member name, so the output is as follows:

```
CallerNameInAttribute.cs:36 - Unspecified member
CallerNameInAttribute.cs:39 - Method
CallerNameInAttribute.cs:40 - Method
CallerNameInAttribute.cs:40 - Method
```

Again, this is in the C# 5 specification, to the extent that it specifies the behavior when the attribute is applied to a function member (method, property, event, and so on) but not to a type. Perhaps it would've been more useful to include types here as well. They're defined to be members of namespaces, so it's not unreasonable for a member name to map to a type name.

Just to reiterate, the reason I included this section was more than for the sake of completeness. It highlights some interesting language choices. When is it okay for

language design to accept limitations to avoid implementation costs? When is it reasonable for a language design choice to conflict with user expectations? When does it make sense for the specification to explicitly turn a decision into an implementation choice? At a meta level, how much time should the language design team spend to specify corner cases for a relatively minor feature? One final piece of practical detail remains before we close the chapter: enabling this feature on frameworks where the attributes don't exist.

7.2.5 Using caller information attributes with old versions of .NET

Hopefully, by now most readers will be targeting .NET 4.5+ or .NET Standard 1.0+, both of which contain the caller information attributes. But in some cases, you're still able to use a modern compiler but need to target old frameworks.

In these cases, you can still use the caller information attributes, but you need to make the attributes available to the compiler. The simplest way of doing this is to use the `Microsoft.Bcl` NuGet package, which provides the attributes and many other features provided by later versions of the framework.

If you can't use the NuGet package for some reason, you can provide the attributes yourself. They're simple attributes with no parameters or properties, so you can copy the declaration directly from the API documentation. They still need to be in the `System.Runtime.CompilerServices` namespace. To avoid type collisions, you'll want to make sure these are available only when the system-provided attributes aren't available. This can be tricky (as all versioning tends to be), and the details are beyond the scope of this book.

When I started writing this chapter, I hadn't expected to write as much about caller information attributes as I ended up with. I can't say I use the feature much in my day-to-day work, but I find the design aspects fascinating. This isn't in spite of it being a minor feature; it's because it's a minor feature. You'd expect major features—dynamic typing, generics, `async/await`—to require significant language design work, but minor features can have all kinds of corner cases, too. Features often interact with each other, so one of the dangers of introducing a new feature is that it might make a future feature harder to design or implement.

Summary

- Captured `foreach` iteration variables are more useful in C# 5.
- You can use caller information attributes to ask the compiler to fill in parameters based on the caller's source file, line number, and member name.
- Caller information attributes demonstrate the level of detail that language design often requires.

Part 3

C# 6

C# 6 is one of my favorite releases. It has lots of features, but they're mostly independent of each other, simple to explain, and easy to apply to existing code. In some ways, they're underwhelming to read about, but they still make a huge difference to the readability of your code. If I ever have to write code in an older version of C#, it's the C# 6 features that I find myself missing most.

Whereas each earlier version of C# introduced a whole new way of thinking about code (generics, LINQ, dynamic typing, and async/await, respectively), C# 6 is more about applying some polish to the code you already have.

I've grouped the features into three chapters: features about properties, features about strings, and features that aren't about properties or strings, but this is somewhat arbitrary. I recommend reading the chapters in the natural order, but there's no big buildup to a grand scheme as there was with LINQ.

Given the way C# 6 features are easily applicable to existing code, I recommend trying them out as you go along. If you maintain a project that has old code you haven't touched in a while, you may find that to be fertile ground for refactoring with the benefit of C# 6.

8

Super-sleek properties and expression-bodied members

This chapter covers

- Implementing read-only properties automatically
- Initializing automatically implemented properties at their declaration
- Removing unnecessary ceremony with expression-bodied members

Some versions of C# have one big, unifying feature that almost all other features contribute to. For example, C# 3 introduced LINQ, and C# 5 introduced asynchrony. C# 6 isn't like that, but it does have a general theme. Almost all the features contribute to cleaner, simpler, and more readable code. C# 6 isn't about doing more; it's about doing the same work with less code.

The features you'll look at in this chapter are about properties and other simple pieces of code. When not much logic is involved, removing even the smallest piece of ceremony—braces and return statements, for example—can make a big

difference. Although the features here may not sound impressive, I've been surprised at their impact in real code. We'll start off looking at properties and move on to methods, indexers, and operators.

8.1 A brief history of properties

C# has had properties from the first version. Although their core functionality hasn't changed over time, they've gradually become simpler to express in source code and more versatile. Properties allow you to differentiate between how state access and manipulation are exposed in the API and how that state is implemented.

For example, suppose you want to represent a point in 2D space. You could represent that easily using public fields, as shown in the following listing.

Listing 8.1 Point class with public fields

```
public sealed class Point
{
    public double X;
    public double Y;
}
```

That doesn't seem too bad at first glance, but the capabilities of the class ("I can access its X and Y values") are closely tied to the implementation ("I'll use two double fields"). But at this point, the implementation has lost control. As long as the class state is exposed directly via fields, you can't do the following:

- Perform validation when setting new values (for example, preventing infinite or not-a-number values for the X and Y coordinates)
- Perform computation when fetching values (for example, if you wanted to store the fields in a different format—unlikely for a point, but perfectly feasible in other cases)

You might argue that you could always change the field to a property later, when you find you need something like this, but that's a breaking change, which you probably want to avoid. (It breaks source compatibility, binary compatibility, and reflection compatibility. That's a big risk to take just to avoid using properties from the start.)

In C# 1, the language provided almost no help with properties. A property-based version of listing 8.1 would require manual declaration of the backing fields, along with getters and setters for each of the properties, as shown in the next listing.

Listing 8.2 Point class with properties in C# 1

```
public sealed class Point
{
    private double x, y;
    public double X { get { return x; } set { x = value; } }
    public double Y { get { return y; } set { y = value; } }
}
```

You could argue that many properties start off simply reading and writing fields with no extra validation, computation, or anything else and stay that way for the whole history of the code. Properties like that could've been exposed as fields, but it's hard to predict which properties might need extra code later. Even when you can do that accurately, it feels like you're operating at two levels of abstraction for no reason. To me, properties act as part of the contract that a type provides: its advertised functionality. Fields are simply implementation details; they're the mechanism inside the box, which users don't need to know about in the vast majority of cases. I prefer fields to be private in almost all cases.

NOTE Like all good rules of thumb, there are exceptions. In some situations, it makes sense to expose fields directly. You'll see one interesting case in chapter 11 when you look at the tuples provided by C# 7.

The only improvement to properties in C# 2 was to allow different access modifiers for the getter and setter—for example, a public getter and a private setter. (That's not the only combination available, but it's by far the most common one.)

C# 3 then added automatically implemented properties, which allow listing 8.2 to be rewritten in a simpler way, as follows.

Listing 8.3 Point class with properties in C# 3

```
public sealed class Point
{
    public double X { get; set; }
    public double Y { get; set; }
}
```

This code is almost exactly equivalent to the code in listing 8.2, except there's no way of accessing the backing fields directly. They're given *unspeakable names*, which aren't valid C# identifiers but are fine as far as the runtime is concerned.

Importantly, C# 3 allowed only read/write properties to be implemented automatically. I'm not going to go into all the benefits (and pitfalls) of immutability here, but there are many reasons you might want your `Point` class to be immutable. To make your properties truly read-only, you need to go back to writing the code manually.

Listing 8.4 Point class with read-only properties via manual implementation in C# 3

```
public sealed class Point
{
    private readonly double x, y; ← Declares
    public double X { get { return x; } } read-only fields
    public double Y { get { return y; } } | Declares read-only properties
                                            | returning the field values

    public Point(double x, double y)
    {
        this.x = x; | Initializes the fields
        this.y = y; | on construction
    }
}
```

This is irritating, to say the least. Many developers—including me—sometimes cheated. If we wanted read-only properties, we'd use automatically implemented properties with private setters, as shown in the following listing.

Listing 8.5 Point class with publicly read-only properties via automatic implementation with private setters in C# 3

```
public sealed class Point
{
    public double X { get; private set; }
    public double Y { get; private set; }

    public Point(double x, double y)
    {
        X = x;
        Y = y;
    }
}
```

That works, but it's unsatisfying. It doesn't express what you want. It allows you to change the values of the properties within the class even though you don't want to; you want a property you can set in the constructor but then never change elsewhere, and you want it to be backed by a field in a trivial way. Up to and including C# 5, the language forced you to choose between simplicity of implementation and clarity of intent, with each choice sacrificing the other. Since C# 6, you no longer need to compromise; you can write brief code that expresses your intent clearly.

8.2 *Upgrades to automatically implemented properties*

C# 6 introduced two new features to automatically implemented properties. Both are simple to explain and use. In the previous section, I focused on the importance of exposing properties instead of public fields and the difficulties of implementing immutable types concisely. You can probably guess how our first new feature in C# 6 works, but a couple of other restrictions have been lifted, too.

8.2.1 *Read-only automatically implemented properties*

C# 6 allows genuinely read-only properties backed by read-only fields to be expressed in a simple way. All it takes is an empty getter and no setter, as shown in the next listing.

Listing 8.6 Point class using read-only automatically implemented properties

```
public sealed class Point
{
    public double X { get; }           | Declares read-only automatically
    public double Y { get; }           | implemented properties

    public Point(double x, double y)
    {
```

```

    X = x;
    Y = y;
}
}                                | Initializes the properties
                                    | on construction

```

The only parts that have changed from listing 8.5 are the declarations of the X and Y properties; they no longer have a setter at all. Given that there are no setters, you may be wondering how you're initializing the properties in the constructor. It happens exactly as it did in listing 8.4, where you implemented it manually: the field declared by the automatically implemented property is read-only, and any assignments to the property are translated by the compiler into direct field assignments. Any attempt to set the property in code other than the constructor results in a compile-time error.

As a fan of immutability, this feels like a real step forward to me. It lets you express your ideal result in a small amount of code. Laziness is now no obstacle to code hygiene, at least in this one small way.

The next limitation removed in C# 6 has to do with initialization. So far, the properties I've shown have either not been initialized explicitly at all or have been initialized in a constructor. But what if you want to initialize a property as if it were a field?

8.2.2 Initializing automatically implemented properties

Before C# 6, any initialization of automatically implemented properties had to be in constructors; you couldn't initialize the properties at the point of declaration. For example, suppose you had a Person class in C# 2, as shown in the following listing.

Listing 8.7 Person class with manual property in C# 2

```

public class Person
{
    private List<Person> friends = new List<Person>();
    public List<Person> Friends
    {
        get { return friends; }
        set { friends = value; }
    }
}

```

Annotations for Listing 8.7:

- A callout points to the declaration of the private field 'friends' with the text "Declares and initializes field".
- A callout points to the declaration of the public property 'Friends' with the text "Exposes a property to read/write the field".

If you wanted to change this code to use automatically implemented properties, you'd have to move the initialization into a constructor, where previously you hadn't explicitly declared any constructors at all. You'd end up with code like the following listing.

Listing 8.8 Person class with automatically implemented property in C# 3

```

public class Person
{
    public List<Person> Friends { get; set; }
    public Person()
    {
}

```

Annotations for Listing 8.8:

- A callout points to the declaration of the automatically implemented property 'Friends' with the text "Declares the property; no initializer permitted".

```

        Friends = new List<Person>();
    }
}

```



Initializes the property
in a constructor

That's about as verbose as it was before! In C# 6, this restriction was removed. You can initialize at the point of property declaration, as the following listing shows.

Listing 8.9 Person class with automatically implemented read/write property in C# 6

```

public class Person
{
    public List<Person> Friends { get; set; } =
        new List<Person>(); // Declares and initializes a read/write
                           // automatically implemented
}

```

Naturally, you can use this feature with read-only automatically implemented properties as well. One common pattern is to have a read-only property exposing a mutable collection, so a caller can add or remove items from the collection but can never change the property to refer to a different collection (or set it to be a null reference). As you might expect, this is just a matter of removing the setter.

Listing 8.10 Person class with automatically implemented read-only property in C# 6

```

public class Person
{
    public List<Person> Friends { get; } =
        new List<Person>(); // Declares and initializes a read-only
                           // automatically implemented
}

```

I've rarely found this particular restriction of earlier versions of C# to be a massive problem, because usually I want to initialize properties based on constructor parameters anyway, but the change is certainly a welcome addition. The next restriction that has been removed ends up being more important in conjunction with read-only automatically implemented properties.

8.2.3 Automatically implemented properties in structs

Before C# 6, I always found automatically implemented properties to be a little problematic in structs. There were two reasons for this:

- I always write immutable structs, so the lack of read-only automatically implemented properties was always a pain point.
- I could assign to an automatically implemented property in a constructor only after chaining to another constructor because of the rules about definite assignment.

NOTE In general, *definite assignment rules* are about the compiler keeping track of which variables will have been assigned at a particular point in your code, regardless of how you got there. These rules are mostly relevant for

local variables, to make sure you don't try to read from a local variable that hasn't been assigned a value yet. Here, we're looking at a slightly different use of the same rules.

The following listing demonstrates both of these points in a struct version of our previous `Point` class. Just typing it out makes me squirm a little.

Listing 8.11 Point struct in C# 5 using automatically implemented properties

```
public struct Point
{
    public double X { get; private set; }
    public double Y { get; private set; }           | Properties with public
                                                    | getters and private setters

    public Point(double x, double y) : this() {      | Chaining to default
        X = x;                                     | constructor
        Y = y;                                     | Property initialization
    }
}
```

This isn't code I would've included in a real codebase. The benefits of automatically implemented properties are outweighed by the ugliness. You're already familiar with the read-only aspect of the properties, but why do you need to call the default constructor in our constructor initializer?

The answer lies in subtleties of the rules around field assignments in structs. Two rules are at work here:

- You can't use any properties, methods, indexers, or events in a struct until the compiler considers that all the fields have been definitely assigned.
- Every struct constructor must assign values to all fields before it returns control to the caller.

In C# 5, without calling the default constructor, you're violating both rules. Setting the `X` and `Y` properties still counts as using the value of the struct, so you're not allowed to do it. Setting the properties doesn't count as assigning the fields, so you can't return from the constructor anyway. Chaining to the default constructor is a workaround because that assigns all fields before your constructor body executes. You can then set the properties and return at the end because the compiler is happy that all your fields were set anyway.

In C# 6, the language and the compiler have a closer understanding of the relationship between automatically implemented properties and the fields they're backed by:

- You're allowed to set an automatically implemented property before all the fields are initialized.
- Setting an automatically implemented property counts as initializing the field.
- You're allowed to read an automatically implemented property before other fields are initialized, so long as you've set it beforehand.

Another way of thinking of this is that within the constructor, automatically implemented properties are treated as if they're fields.

With those new rules in place and genuine read-only automatically implemented properties, the struct version of `Point` in C# 6 shown in the next listing is identical to the class version in listing 8.6, other than declaring a struct instead of a sealed class.

Listing 8.12 Point struct in C# 6 using automatically implemented properties

```
public struct Point
{
    public double X { get; }
    public double Y { get; }

    public Point(double x, double y)
    {
        X = x;
        Y = y;
    }
}
```

The result is clean and concise, just the way you want it.

NOTE You may be asking whether `Point` should be a struct at all. In this case, I'm on the fence. Points do feel like fairly natural value types, but I still usually default to creating classes. Outside Noda Time (which is struct heavy), I rarely write my own structs. This example certainly isn't trying to suggest you should start using structs more widely, but if you do write your own struct, the language is more helpful than it used to be.

Everything you've seen so far has made automatically implemented properties cleaner to work with, which often reduces the amount of boilerplate code. Not all properties are automatically implemented, though. The mission of removing clutter from your code doesn't stop there.

8.3 Expression-bodied members

Far be it from me to prescribe one specific style of coding in C#. Aside from anything else, different problem domains lend themselves to different approaches. But I've certainly come across types that have a lot of simple methods and properties. C# 6 helps you here with *expression-bodied members*. We'll start off with properties, since you were looking at them in the previous section, and then see how the same idea can be applied to other function members.

8.3.1 Even simpler read-only computed properties

Some properties are trivial: if the implementation in terms of fields matches the logical state of the type, the property can return the field value directly. That's what automatically implemented properties are for. Other properties involve computations based on other fields or properties. To demonstrate the problem that C# 6 addresses,

the following listing adds another property to our `Point` class: `DistanceFromOrigin`, which uses the Pythagorean theorem in a simple way to return how far the point is from the origin.

NOTE Don't worry if the math here isn't familiar. The details aren't important, just the fact that it's a read-only property that uses `X` and `Y`.

Listing 8.13 Adding a `DistanceFromOrigin` property to `Point`

```
public sealed class Point
{
    public double X { get; }
    public double Y { get; }

    public Point(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double DistanceFromOrigin
    {
        get { return Math.Sqrt(X * X + Y * Y); }
    }
}
```

Read-only property to
compute a distance

I'm not going to claim that this is terribly hard to read, but it does contain a lot of syntax that I could describe as *ceremony*: it's there only to make the compiler aware of how the meaningful code fits in. Figure 8.1 shows the same property but annotated to highlight the useful parts; the ceremony (braces, a return statement, and a semicolon) are in a lighter shade.

C# 6 allows you to express this much more cleanly:

```
public double DistanceFromOrigin => Math.Sqrt(X * X + Y * Y);
```

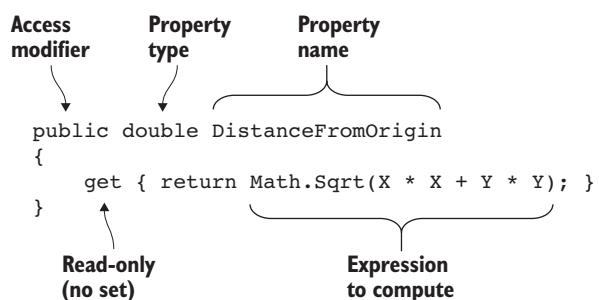


Figure 8.1 Annotated property declaration showing important aspects

Here, the `=>` is used to indicate an *expression-bodied member*—in this case, a read-only property. No more braces, no more keywords. Both the read-only property part and the fact that the expression is used to return the value are implicit. Compare this with figure 8.1, and you'll see that the expression-bodied form has everything that's useful (with a different way of indicating that it's a read-only property) and nothing extraneous. Perfect!

No, this isn't a lambda expression

Yes, you've seen this element of syntax before. Lambda expressions were introduced in C# 3 as a brief way of declaring delegates and expression trees. For example:

```
Func<string, int> stringLength = text => text.Length;
```

Expression-bodied members use the `=>` syntax but aren't lambda expressions. The preceding declaration of `DistanceFromOrigin` doesn't involve any delegates or expression trees; it only instructs the compiler to create a read-only property that computes the given expression and returns the result.

When talking about the syntax out loud, I usually describe `=>` as a *fat arrow*.

You may be wondering whether this is useful in the real world rather than just in made-up examples for the book. To show you concrete examples, I'll use Noda Time.

PASS-THROUGH OR DELEGATING PROPERTIES

We'll briefly consider three types from Noda Time:

- **LocalDate**—Just a date in a particular calendar with no time component
 - **LocalTime**—A time of day with no date component
 - **LocalDateTime**—The combination of a date and a time

Don't worry about the details of initialization and so on; just think about what you'd want out of the three types. Obviously, a date will have properties for the year, month, and day, and a time will have hours, minutes, seconds, and so on. What about the type combining the two? It's handy to be able to get the date and time components separately, but often you want the subcomponents of the date and time. Each implementation of `LocalDate` and `LocalTime` is carefully optimized, and I wouldn't want to duplicate that logic in `LocalDateTime`, so the subcomponent properties are pass-throughs that delegate to properties of the date or time components. The implementation shown in the following listing is now extremely clean.

Listing 8.14 Delegating properties in Noda Time

```
public struct LocalDateTime
{
    public LocalDate Date { get; } ← Property for the date component
    public int Year => Date.Year;
    public int Month => Date.Month;
    public int Day => Date.Day;

    public LocalTime TimeOfDay { get; } ← Properties delegating to date subcomponents
    public int Hour => TimeOfDay.Hour;
    public int Minute => TimeOfDay.Minute;
    public int Second => TimeOfDay.Second; ← Property for the time component

    } ← Initialization, other properties, and members
```

A lot of properties are like this; removing the `{ get { return ... } }` part from each of them was a real pleasure and leaves the code much clearer.

PERFORMING SIMPLE LOGIC ON ANOTHER PIECE OF STATE

Within `LocalTime`, there's a single piece of state: the nanosecond within the day. All the other properties compute a value based on that. For example, the code to compute the subsecond value in nanoseconds is a simple remainder operation:

```
public int NanosecondOfSecond =>
    (int) (NanosecondOfDay % NodaConstants.NanosecondsPerSecond);
```

That code will get even simpler in chapter 10, but for now, you can just enjoy the brevity of the expression-bodied property.

Important caveat

Expression-bodied properties have one downside: there's only a single-character difference between a read-only property and a public read/write field. In most cases, if you make a mistake, a compile-time error will occur, due to using other properties or fields within a field initializer, but for static properties or properties returning a constant value, it's an easy mistake to make. Consider the difference between the following declarations:

```
// Declares a read-only property
public int Foo => 0;
// Declares a read/write public field
public int Foo = 0;
```

This has been a problem for me a couple of times, but after you're aware of it, checking for it is easy enough. Make sure your code reviewers are aware of it, too, and you're unlikely to get caught.

So far, we've concentrated on properties as a natural segue from the other new property-related features. As you may have guessed from the section title, however, other kinds of members can have expression bodies.

8.3.2 Expression-bodied methods, indexers, and operators

In addition to expression-bodied properties, you can write expression-bodied methods, read-only indexers, and operators, including user-defined conversions. The `=>` is used in the same way, with no braces surrounding the expression and no explicit return statement.

For example, a simple `Add` method and its operator equivalent to add a `Vector` (with obvious `X` and `Y` properties) to a `Point` might look like the following listing in C# 5.

Listing 8.15 Simple methods and operators in C# 5

```
public static Point Add(Point left, Vector right)
{
    return left + right;
```

Just delegate to
the operator.

```

    }

public static Point operator+(Point left, Vector right)
{
    return new Point(left.X + right.X,
                    left.Y + right.Y);
}

```

**Simple constructor
call to implement +**

In C# 6, it could look simpler, with both of these being implemented using expression-bodied members, as in the next listing.

Listing 8.16 Expression-bodied methods and operators in C# 6

```

public static Point Add(Point left, Vector right) => left + right;

public static Point operator+(Point left, Vector right) =>
    new Point(left.X + right.X, left.Y + right.Y);

```

Note the formatting I've used in `operator+;` putting everything on one line would make it much too long. In general, I put the `=>` at the end of the declaration part and indent the body as usual. The way you format your code is entirely up to you, but I've found this convention works well for all kinds of expression-bodied members.

You can also use expression bodies for void-returning methods. In that case, there's no `return` statement to omit; only the braces are removed.

NOTE This is consistent with the way lambda expressions work. As a reminder, expression-bodied members aren't lambda expressions, but they have this aspect in common.

For example, consider a simple log method:

```

public static void Log(string text)
{
    Console.WriteLine("{0:o}: {1}", DateTime.UtcNow, text)
}

```

This could be written with an expression-bodied method like this instead:

```

public static void Log(string text) =>
    Console.WriteLine("{0:o}: {1}", DateTime.UtcNow, text);

```

Here the benefit is definitely smaller, but for methods where the declaration and body fit on one line, it can still be worth doing. In chapter 9, you'll see a way of making this even cleaner using interpolated string literals.

For a final example with methods, a property, and an indexer, let's imagine you want to create your own `IReadOnlyList<T>` implementation to provide a read-only view over any `IList<T>`. Of course, `ReadOnlyCollection<T>` already does this, but it also implements the mutable interfaces (`IList<T>, ICollection<T>`). At times you may want to be precise about what a collection allows via the interfaces it implements. With expression-bodied members, the implementation of such a wrapper is short indeed.

Listing 8.17 `IReadOnlyList<T>` implementation using expression-bodied members

```
public sealed class ReadOnlyListView<T> : IReadOnlyList<T>
{
    private readonly IList<T> list;

    public ReadOnlyListView(IList<T> list)
    {
        this.list = list;
    }

    public T this[int index] => list[index];           ← Indexer delegating
    public int Count => list.Count;                    ← to list indexer
    public IEnumerator<T> GetEnumerator() =>          ← Property delegating
        list.GetEnumerator();                         to list property
    Ienumerator Ienumerable.GetEnumerator() =>      ← Method delegating
        GetEnumerator();                           to list method
    }                                              ← Method delegating to the
                                                other GetEnumerator method
}
```

The only new feature shown here is the syntax for expression-bodied indexers, and I hope it's sufficiently similar to the syntax for the other kinds of members that you didn't even notice it was new.

Does anything stick out to you, though? Anything surprise you at all? That constructor looks a little ugly, doesn't it?

8.3.3 Restrictions on expression-bodied members in C# 6

Normally, at this point, having just remarked on how verbose a piece of code is, I'd reveal the good news of another feature that C# has implemented to make it better. Not this time, I'm afraid—at least not in C# 6.

Even though the constructor has only a single statement, there's no such thing as an expression-bodied constructor in C# 6. It's not alone, either. You can't have expression-bodied

- Static constructors
- Finalizers
- Instance constructors
- Read/write or write-only properties
- Read/write or write-only indexers
- Events

None of that keeps me awake at night, but the inconsistency apparently bothered the C# team enough that C# 7 allows all of these to be expression-bodied. They don't typically save any printable characters, but formatting conventions allow them to save vertical space, and there's still the readability hint that this is just a simple member. They all use the same syntax you're already used to, and listing 8.18 gives a complete example, purely for the sake of showing the syntax. This code isn't intended to be useful other than as an example, and in the case of the event handler, it's dangerously non-thread-safe compared with a simple field-like event.

Listing 8.18 Extra expression-bodied members in C# 7

```

public class Demo
{
    static Demo() =>
        Console.WriteLine("Static constructor called");
    ~Demo() => Console.WriteLine("Finalizer called");
    private string name;
    private readonly int[] values = new int[10];
    public Demo(string name) => this.name = name;
    private PropertyChangedEventHandler handler;
    public event PropertyChangedEventHandler PropertyChanged
    {
        add => handler += value;
        remove => handler -= value;
    }
    public int this[int index]
    {
        get => values[index];
        set => values[index] = value;
    }
    public string Name
    {
        get => name;
        set => name = value;
    }
}

```

One nice aspect of this is that the `get` accessor can be expression-bodied even if the `set` accessor isn't, or vice versa. For example, suppose you want to make your indexer setter validate that the new value isn't negative. You could still keep an expression-bodied getter:

```

public int this[int index]
{
    get => values[index];
    set
    {
        if (value < 0)
        {
            throw new ArgumentOutOfRangeException();
        }
        Values[index] = value;
    }
}

```

I expect this to be reasonably common in the future. Setters tend to have validation, whereas getters are usually trivial, in my experience.

TIP If you find yourself writing a lot of logic in a getter, it's worth considering whether it should be a method. Sometimes the boundary can be fuzzy.

With all the benefits of expression-bodied members, do they have any other downsides? How aggressive should you be in converting everything you possibly can to use them?

8.3.4 Guidelines for using expression-bodied members

My experience is that expression-bodied members are particularly useful for operators, conversions, comparisons, equality checks, and `ToString` methods. These usually consist of simple code, but for some types there can be an awful lot of these members, and the difference in readability can be significant.

Unlike some features that are somewhat niche, expression-bodied members can be used to significant effect in pretty much every codebase I've come across. When I converted Noda Time to use C# 6, I removed roughly 50% of the return statements in the code. That's a huge difference, and it'll only increase as I gradually take advantage of the extra opportunities afforded by C# 7.

There's more to expression-bodied members than readability, mind you. I've found that they provide a psychological effect: it feels like I'm doing functional programming to a greater extent than before. That, in turn, makes me feel smarter. Yes, that's as silly as it sounds, but it really does feel satisfying. You may be more rational than me, of course.

The danger, as always, is overuse. In some cases, you can't use expression-bodied members, because your code includes a `for` statement or something similar. In plenty of cases, it's possible to convert a regular method into an expression-bodied member, but you really shouldn't. I've found that there are two categories of members like this:

- Members performing precondition checks
- Members using explanatory variables

As an example of the first category, I have a class called `Preconditions` with a generic `CheckNotNull` method that accepts a reference and a parameter name. If the reference is null, it throws an `ArgumentNullException` using the parameter name; otherwise, it returns the value. This allows a convenient combination of check and assign statements in constructors and the like.

This also allows—but certainly doesn't force—you to use the result as both the target of a method call or, indeed, an argument to it. The problem is, understanding what's going on becomes difficult if you're not careful. Here's a method from the `LocalDateTime` struct I described earlier:

```
public ZonedDateTime InZone(
    DateTimeZone zone,
    ZoneLocalMappingResolver resolver)
{
    Preconditions.CheckNotNull(zone);
    Preconditions.CheckNotNull(resolver);
    return zone.ResolveLocal(this, resolver);
}
```

This reads nice and simply: check that the arguments are valid and then do the work by delegating to another method. This could be written as an expression-bodied member, like this:

```
public ZonedDateTime InZone(
    DateTimeZone zone,
    ZoneLocalMappingResolver resolver) =>
    Preconditions.checkNotNull(zone)
        .ResolveLocal(
            this,
            Preconditions.checkNotNull(resolver));
```

That would have exactly the same effect, but it's much harder to read. In my experience, one validation check puts a method on the borderline for expression-bodied members; with two of them, it's just too painful.

For explanatory variables, the `NanosecondOfDay` example I provided earlier is just one of many properties on `LocalTime`. About half of them use expression bodies, but quite a few of them have two statements, like this:

```
public int Minute
{
    get
    {
        int minuteOfDay = (int) NanosecondOfDay / NanosecondsPerMinute;
        return minuteOfDay % MinutesPerHour;
    }
}
```

That can easily be written as an expression-bodied property by effectively inlining the `minuteOfDay` variable:

```
public int Minute =>
    ((int) NanosecondOfDay / NodaConstants.NanosecondsPerMinute) %
        NodaConstants.MinutesPerHour;
```

Again, the code achieves exactly the same goal, but in the original version, the `minuteOfDay` variable adds information about the *meaning* of the subexpression, making the code easier to read.

On any given day, I *might* come to a different conclusion. But in more complex cases, following a sequence of steps and naming the results can make all the difference when you come back to the code six months later. It also helps you if you ever need to step through the code in a debugger, as you can easily execute one statement at a time and check that the results are the ones you expect.

The good news is that you can experiment and change your mind as often as you like. Expression-bodied members are purely syntactic sugar, so if your taste changes over time, you can always convert more code to use them or revert code that used expression bodies a little too eagerly.

Summary

- Automatically implemented properties can now be read-only and backed by a read-only field.
- Automatically implemented properties can now have initializers rather than nondefault values having to be initialized in a constructor.
- Structs can use automatically implemented properties without having to chain constructors together.
- Expression-bodied members allow simple (single-expression) code to be written with less ceremony.
- Although restrictions limit the kinds of members that can be written with expression bodies in C# 6, those restrictions are lifted in C# 7.



Stringy features

This chapter covers

- Using interpolated string literals for more-readable formatting
- Working with `FormattableString` for localization and custom formatting
- Using `nameof` for refactoring-friendly references

Everyone knows how to use strings. If `string` isn't the first .NET data type you learned about, it's probably the second. The `string` class itself hasn't changed much over the course of .NET's history, and not many string-oriented features have been introduced in C# as a language since C# 1. C# 6, however, changed that with another kind of string literal and a new operator. You'll look at both of these in detail in this chapter, but it's worth remembering that the strings themselves haven't changed at all. Both features provide new ways of *obtaining* strings, but that's all.

Just like the features you saw in chapter 8, string interpolation doesn't allow you to do anything you couldn't do before; it just allows you to do it more readably and concisely. That's not to diminish the importance of the feature. Anything that

allows you to write clearer code more quickly—and then read it more quickly later—will make you more productive.

The `nameof` operator was genuinely new functionality in C# 6, but it's a reasonably minor feature. All it does is allow you to get an identifier that already appears in your code but as a string at execution time. It's not going to change your world like LINQ or `async/await`, but it helps avoid typos and allows refactoring tools to do more work for you. Before I show you anything new, let's revisit what you already know.

9.1 A recap on string formatting in .NET

You almost certainly know everything in this section. You may well have been using strings for many years and almost certainly for as long as you've been using C#. Still, in order to understand how the interpolated string literal feature in C# 6 works, it's best to have all that knowledge uppermost in your mind. Please bear with me as we go over the basics of how .NET handles string formatting. I promise we'll get to the new stuff soon.

9.1.1 Simple string formatting

If you're like me, you like experimenting with new languages by writing trivial console applications that do nothing useful but give the confidence and firm foundation to move on to more-impressive feats. As such, I can't remember how many languages I've used to implement the functionality shown next—asking the user's name and then saying hello to that user:

```
Console.Write("What's your name? ");
string name = Console.ReadLine();
Console.WriteLine("Hello, {0}!", name);
```

The last line is the most relevant one for this chapter. It uses an overload of `Console.WriteLine`, which accepts a *composite format string* including *format items* and then arguments to replace those format items. The preceding example has one format item, `{0}`, which is replaced by the value of the `name` variable. The number in the format item specifies the index of the argument you want to fill the hole (where 0 represents the first of the values, 1 represents the second, and so on).

This pattern is used in various APIs. The most obvious example is the static `Format` method in the `string` class, which does nothing *but* format the string appropriately. So far, so good. Let's do something a little more complicated.

9.1.2 Custom formatting with format strings

Just to be clear, my motivation for including this subsection is as much for my future self as for you, dear reader. If MSDN displayed the number of times I've visited any given page, the number for the page on composite format strings would be frightening. I keep forgetting exactly what goes where and what terms to use, and I figured

that if I included that information here, I might start remembering it better. I hope you find it helpful in the same way.

Each format item in a composite format string specifies the index of the argument to be formatted, but it can also specify the following options for formatting the value:

- An *alignment*, which specifies a minimum width and whether the value should be left- or right-aligned. Right-alignment is indicated by a positive value; left-alignment is indicated by a negative value.
- A *format string* for the value. This is probably used most often for date and time values or numbers. For example, to format a date according to ISO-8601, you could use a format string of yyyy-MM-dd. To format a number as a currency value, you could use a format string of c. The meaning of the format string depends on the type of value being formatted, so you need to look up the relevant documentation to choose the right format string.

Figure 9.1 shows all the parts of a composite format string you could use to display a price.

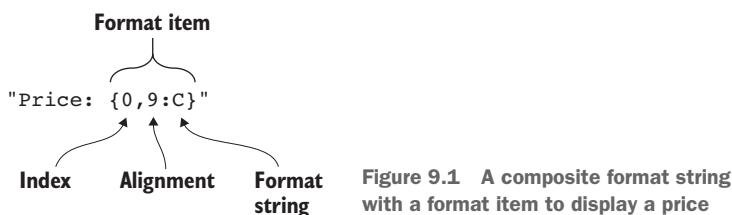


Figure 9.1 A composite format string with a format item to display a price

The alignment and the format string are independently optional; you can specify either, both, or neither. A comma in the format item indicates an alignment, and a colon indicates a format string. If you need a comma in the format string, that's fine; there's no concept of a second alignment value.

As a concrete example to expand on later, let's use the code from figure 9.1 in a broader context, showing different lengths of results to demonstrate the point of alignment. Listing 9.1 displays a price (\$95.25), tip (\$19.05), and total (\$114.30), lining up the labels on the left and the values on the right.

The output on a machine using the US English culture settings by default, would look like this:

```
Price:      $95.25
Tip:       $19.05
Total:     $114.30
```

To make the values right-aligned (or left-padded with spaces, to look at it the other way around), the code uses an alignment value of 9. If you had a huge bill (a million dollars, for example), the alignment would have no effect; it specifies only a minimum width. If you wanted to write code that right-aligned every possible set of values, you'd

have to work out how wide the biggest one would be first. That's pretty unpleasant code, and I'm afraid nothing in C# 6 makes it easier.

Listing 9.1 Displaying a price, tip, and total with values aligned

```
decimal price = 95.25m;
decimal tip = price * 0.2m;                                ←
Console.WriteLine("Price: {0,9:C}", price);
Console.WriteLine("Tip:   {0,9:C}", tip);                      20% tip
Console.WriteLine("Total: {0,9:C}", price + tip);
```

When I showed the output of listing 9.1 on a machine in the US English culture, the part about the culture was important. On a machine using a UK English culture, the code would use £ signs instead. On a machine in the French culture, the decimal separator would become a comma, the currency sign would become a Euro symbol, and that symbol would be at the end of the string instead of the start! Such are the joys of localization, which you'll look at next.

9.1.3 Localization

In broad terms, *localization* is the task of making sure your code does the right thing for all your users, no matter where they are in the world. Anyone who claims that localization is simple is either much more experienced at it than I am or hasn't done enough of it to see how painful it can be. Considering the world is basically round, it certainly seems to have a lot of nasty corner cases to handle. Localization is a pain in all programming languages, but each has a slightly different way of addressing the problems.

NOTE Although I use the term *localization* in this section, other people may prefer the term *globalization*. Microsoft uses the two terms in a slightly different way than other industry bodies, and the difference is somewhat subtle. Experts, please forgive the hand-waving here; the big picture is more important than the fine details of terminology, just this once.

In .NET, the most important type to know about for localization purposes is `CultureInfo`. This is responsible for the cultural preferences of a language (such as English), or a language in a particular location (such as French in Canada), or a particular variant of a language in a location (such as simplified Chinese as used in Taiwan). These cultural preferences include various translations (the words used for the days of the week, for example) and indicate how text is sorted and how numbers are formatted (whether to use a period or comma as the decimal separator) and much more.

Often, you won't see `CultureInfo` in a method signature, but instead the `IFormatProvider` interface, which `CultureInfo` implements. Most formatting methods have overloads with an `IFormatProvider` as the first parameter before the format string itself. For example, consider these two signatures from `string.Format`:

```
static string Format(IFormatProvider provider,
                     string format, params object[] args)
static string Format(string format, params object[] args)
```

Usually, if you provide overloads that differ only by a single parameter, that parameter is the last one, so you might have expected the provider parameter to come after args. That wouldn't work, though, because args is a parameter array (it uses the params modifier). If a method has a parameter array, that has to be the final parameter.

Even though the parameter is of type `IFormatProvider`, the value you pass in as an argument is almost always `CultureInfo`. For example, if you want to format my date of birth for US English—June 19, 1976—you could use this code:

```
var usEnglish = CultureInfo.GetCultureInfo("en-US");
var birthDate = new DateTime(1976, 6, 19);
string formatted = string.Format(usEnglish, "Jon was born on {0:d}", birthDate);
```

Here, `d` is the standard date/time format specifier for *short date*, which in US English corresponds to month/day/year. My date of birth would be formatted as 6/19/1976, for example. In British English, the short date format is day/month/year, so the same date would be formatted as 19/06/1976. Notice that not just the ordering is different: the month is 0-padded to two digits in the British formatting, too.

Other cultures can use entirely different formatting. It can be instructive to see just how different the results of formatting the same value can be between cultures. For example, you could format the same date in every culture .NET knows about as shown in the next listing.

Listing 9.2 Formatting a single date in every culture

```
var cultures = CultureInfo.GetCultures(CultureTypes.AllCultures);
var birthDate = new DateTime(1976, 6, 19);
foreach (var culture in cultures)
{
    string text = string.Format(
        culture, "{0,-15} {1,12:d}", culture.Name, birthDate);
    Console.WriteLine(text);
}
```

The output for Thailand shows that I was born in 2519 in the Thai Buddhist calendar, and the output for Afghanistan shows that I was born in 1355 in the Islamic calendar:

```
...
tg-Cyrl      19.06.1976
tg-Cyrl-TJ   19.06.1976
th          19/6/2519
th-TH       19/6/2519
ti          19/06/1976
ti-ER       19/06/1976
...
ur-PK       19/06/1976
uz          19/06/1976
uz-Arab     29/03 1355
uz-Arab-AF  29/03 1355
uz-Cyrl    19/06/1976
uz-Cyrl-UZ 19/06/1976
...
```

This example also shows a negative alignment value to left-align the culture name using the `{0,-15}` format item while keeping the date right-aligned with the `{1,12:d}` format item.

FORMATTING WITH THE DEFAULT CULTURE

If you don't specify a format provider, or if you pass null as the argument corresponding to an `IFormatProvider` parameter, `CultureInfo.CurrentCulture` will be used as a default. Exactly what that means will depend on your context; it can be set on a per thread basis, and some web frameworks will set it before processing a request on a particular thread.

All I can advise about using the default is to be careful: make sure you know that the value in your specific thread will be appropriate. (Checking the exact behavior is particularly worthwhile if you start parallelizing operations across multiple threads, for example.) If you don't want to rely on the default culture, you'll need to know the culture of the end user you need to format the text for and do so explicitly.

FORMATTING FOR MACHINES

So far, we've assumed that you're trying to format the text for an end user. But that's often not the case. For machine-to-machine communication (such as in URL query parameters to be parsed by a web service), you should use the *invariant culture*, which is obtained via the static `CultureInfo.InvariantCulture` property.

For example, suppose you were using a web service to fetch the list of best sellers from a publisher. The web service might use a URL of <https://manning.com/webservices/bestsellers> but allow a query parameter called `date` to allow you to find out the best-selling books on a particular date.¹ I'd expect that query parameter to use an ISO-8601 format (year first, using dashes between the year, month, and day) for the date. For example, if you wanted to retrieve the best-selling books as of the start of March 20, 2017, you'd want to use a URL of <https://manning.com/webservices/bestsellers?date=2017-03-20>. To construct that URL in code in an application that allows the user to pick a specific date, you might write something like this:

```
string url = string.Format(
    CultureInfo.InvariantCulture,
    "{0}?date={1:yyyy-MM-dd}",
    webServiceBaseUrl,
    searchDate);
```

Most of the time, you shouldn't be directly formatting data for machine-to-machine communication yourself, mind you. I advise you to avoid string conversions wherever you can; they're often a code smell showing that either you're not using a library or framework properly or that you have data design issues (such as storing dates in a database as text instead of as a native date/time type). Having said that, you may well find yourself building strings manually like this more often than you'd like; just pay attention to which culture you should be using.

¹ This is a fictional web service, as far as I'm aware.

Okay, that was a long introduction. But with all this formatting information buzzing around your brain and somewhat ugly examples niggling at you, you're in the right frame of mind to welcome interpolated string literals in C# 6. All those calls to `string.Format` look unnecessarily long-winded, and it's annoying having to look between the format string and the argument list to see what will go where. Surely, we can make our code clearer than that.

9.2 *Introducing interpolated string literals*

Interpolated string literals in C# 6 allow you to perform all this formatting in a much simpler way. The concepts of a format string and arguments still apply, but with interpolated string literals, you specify the values and their formatting information inline, which leads to code that's much easier to read. If you look through your code and find a lot of calls to `string.Format` using hardcoded format strings, you'll love interpolated string literals.

String interpolation isn't a new idea. It's been in many programming languages for a long time, but I've never felt it to be as neatly integrated as it is in C#. That's particularly remarkable when you consider that adding a feature into a language when it's already mature is harder than building it into the first version.

In this section, you'll look at some simple examples before exploring interpolated verbatim string literals. You'll learn how localization can be applied using `FormattableString` and then take a closer look at how the compiler handles interpolated string literals. We'll round off the section with discussion about where this feature is most useful as well as its limitations.

9.2.1 *Simple interpolation*

The simplest way to demonstrate interpolated string literals in C# 6 is to show you the equivalent to the earlier example in which we asked for the user's name. The code doesn't look hugely different; in particular, only the last line has changed at all.

C# 5—old-style style formatting	C# 6—interpolated string literal
<pre>Console.Write("What's your name? "); string name = Console.ReadLine(); Console.WriteLine("Hello, {0}!", name);</pre>	<pre>Console.Write("What's your name? "); string name = Console.ReadLine(); Console.WriteLine(\$"Hello, {name}!");</pre>

The interpolated string literal is shown in bold. It starts with a \$ before the opening double quote; that's what makes it an interpolated string literal rather than a regular one, as far as the compiler is concerned. It contains `{name}` instead of `{0}` for the format item. The text in the braces is an expression that's evaluated and then formatted within the string. Because you've provided all the information you need, the second argument to `WriteLine` isn't required anymore.

NOTE I've lied a little here, for the sake of simplicity. This code doesn't work quite the same way as the original code. The original code passed all the arguments to the appropriate `Console.WriteLine` overload, which performed the formatting for you. Now, all the formatting is performed with a `string.Format` call, and then the `Console.WriteLine` call uses the overload, which has just a string parameter. The result will be the same, though.

Just as with expression-bodied members, this doesn't look like a huge improvement. For a single format item, the original code doesn't have a lot to be confused by. The first couple of times you see this, it might even take you a little longer to read an interpolated string literal than a string formatting call. I was skeptical about just how much I'd ever like them, but now I often find myself converting pieces of old code to use them almost automatically, and I find the readability improvement is often significant.

Now that you've seen the simplest example, let's do something a bit more complex. You'll follow the same sequence as before, first looking at controlling the formatting of values more carefully and then considering localization.

9.2.2 Format strings in interpolated string literals

Good news! There's nothing new to learn here. If you want to provide an alignment or a format string with an interpolated string literal, you do it the same way you would in a normal composite format string: you add a comma before the alignment and a colon before the format string. Our earlier composite formatting example changes in the obvious way, as shown in the following listing.

Listing 9.3 Aligned values using interpolated string literals

```
decimal price = 95.25m;
decimal tip = price * 0.2m;           ← 20% tip
Console.WriteLine($"Price: {price,9:C}");
Console.WriteLine($"Tip: {tip,9:C}");
Console.WriteLine($"Total: {price + tip,9:C}"); | Right-justify prices using
                                                | nine-digit alignment
```

Note that in the last line, the interpolated string doesn't just contain a simple variable for the argument; it performs the addition of the tip to the price. The expression can be any expression that computes a value. (You can't just call a method with a `void` return type, for example.) If the value implements the `IFormatable` interface, its `ToString(string, IFormatProvider)` method will be called; otherwise, `System.Object.ToString()` is used.

9.2.3 Interpolated verbatim string literals

You've no doubt seen *verbatim string literals* before; they start with @ before the double quote. Within a verbatim string literal, backslashes and line breaks are included in the string. For example, in the verbatim string literal `@"c:\\Windows"`, the backslash really is a backslash; it isn't the start of an escape sequence. The only kind of escape sequence within a verbatim string literal is when you have two double quote characters together,

which results in one double quote character in the resulting string. Verbatim string literals are typically used for the following:

- Strings breaking over multiple lines
- Regular expressions (which use backslashes for escaping, quite separate from the escaping the C# compiler uses in regular string literals)
- Hardcoded Windows filenames

NOTE With multiline strings, you should be careful about exactly which characters end up in your string. Although the difference between “carriage-return” and “carriage-return line-feed separators” is irrelevant in most code, it’s significant in verbatim string literals.

The following shows a quick example of each of these:

```
string sql = @"
    SELECT City, ZipCode
    FROM Address
    WHERE Country = 'US'";
Regex lettersDotDigits = new Regex(@"[a-z]+\.\d+");
string file = @"c:\users\skeet\Test\Test.cs"
```

Verbatim string literals can be interpolated as well; you put a \$ in front of the @, just as you would to interpolate a regular string literal. Our earlier multiline output could be written using a single interpolated verbatim string literal, as shown in the following listing.

Listing 9.4 Aligned values using a single interpolated verbatim string literal

```
decimal price = 95.25m;
decimal tip = price * 0.2m;
Console.WriteLine($@"Price: {price,9:C}           ↪ 20% tip
Tip:   {tip,9:C}
Total: {price + tip,9:C}");
```

I probably *wouldn’t* do this; it’s just not as clean as using three separate statements. I’m using the preceding code only as a simple example of what’s possible. Consider it for places where you’re already using verbatim string literals sensibly.

TIP The order of the symbols matters. `$@"Text"` is a valid interpolated verbatim string literal, but `@$"Text"` isn’t. I admit I haven’t found a good mnemonic device to remember this. Just try whichever way you think is right, and change it if the compiler complains!

This is all very convenient, but I’ve shown only the surface level of what’s going on. I’ll assume you bought this book because you want to know about the features inside and out.

9.2.4 Compiler handling of interpolated string literals (part 1)

The compiler transformation here is simple. It converts the interpolated string literal into a call to `string.Format`, and it extracts the expressions from the format items and passes them as arguments after the composite format string. The expression is replaced with the appropriate index, so the first format item becomes `{0}`, the second becomes `{1}`, and so on.

To make this clearer, let's consider a trivial example, this time separating the formatting from the output for clarity:

```
int x = 10;
int y = 20;
string text = $"x={x}, y={y}";
Console.WriteLine(text);
```

This is handled by the compiler as if you'd written the following code instead:

```
int x = 10;
int y = 20;
string text = string.Format("x={0}, y={1}", x, y);
Console.WriteLine(text);
```

The transformation is that simple. If you want to go deeper and verify it for yourself, you could use a tool such as ildasm to look at the IL that the compiler has generated.

One side effect of this transformation is that unlike regular or verbatim string literals, interpolated string literals don't count as constant expressions. Although in some cases the compiler could reasonably consider them to be constant (if they don't have any format items or if all the format items are just string constants without any alignment or format strings), these would be corner cases that would complicate the language for little benefit.

So far, all our interpolated strings have resulted in a call to `string.Format`. That doesn't always happen, though, and for good reasons, as you'll see in the next section.

9.3 Localization using *FormattableString*

In section 9.1.3, I demonstrated how string formatting can take advantage of different format providers—typically using `CultureInfo`—to perform localization. All the interpolated string literals you've seen so far would've been evaluated using the default culture for the executing thread, so our price examples in 9.1.2 and 9.2.2 could easily have different output on your machine than the result I showed.

To perform formatting in a specific culture, you need three pieces of information:

- The composite format string, which includes the hardcoded text and the format items as placeholders for the real values
- The values themselves
- The culture you want to format the string in

You can slightly rewrite our first example of formatting in a culture to store each of these in a separate variable, and then call `string.Format` at the end:

```
var compositeFormatString = "Jon was born on {0:d}";
var value = new DateTime(1976, 6, 19);
var culture = CultureInfo.GetCultureInfo("en-US");
var result = string.Format(culture, compositeFormatString, value);
```

How can you do this with interpolated string literals? An interpolated string literal contains the first two pieces of information (the composite format string and the values to format), but there's nowhere to put the culture. That would be fine if you could get at the individual pieces of information afterward, but every use of interpolated string literals that you've seen so far has performed the string formatting as well, leaving you with just a single string as the result.

That's where `FormattableString` comes in. This is a class in the `System` namespace introduced in .NET 4.6 (and .NET Standard 1.3 in the .NET Core world). It holds the composite format string and the values so they can be formatted in whatever culture you want later. The compiler is aware of `FormattableString` and can convert an interpolated string literal into a `FormattableString` instead of a string where necessary. That allows you to rewrite our simple date-of-birth example as follows:

```
var dateOfBirth = new DateTime(1976, 6, 19);
FormattableString formattableString =
    $"Jon was born on {dateOfBirth:d}";           ← Keeps the composite format string
                                                and value in a FormattableString
var culture = CultureInfo.GetCultureInfo("en-US");
var result = formattableString.ToString(culture); ← Formats in the
                                                specified culture
```

Now that you know the basic reason for the existence of `FormattableString`, you can look at how the compiler uses it and then examine localization in more detail. Although localization is certainly the primary motivation for `FormattableString`, it can be used in other situations as well, which you'll look at in section 9.3.3. The section then concludes with your options if your code is targeting an earlier version of .NET.

9.3.1 ***Compiler handling of interpolated string literals (part 2)***

In a reversal of my earlier approach, this time it makes sense to talk about how the compiler considers `FormattableString` before moving on to examining its uses in detail. The compile-time type of an interpolated string literal is `string`. There's no conversion from `string` to `FormattableString` or to `IFormattable` (which `FormattableString` implements), but there are conversions from interpolated string literal expressions to both `FormattableString` and `IFormattable`.

The differences between conversions from an expression to a type and conversions from a type to another type are somewhat subtle, but this is nothing new. For example, consider the integer literal 5. Its type is `int`, so if you declare `var x = 5`, the type of `x` will be `int`, but you can also use it to initialize a variable of type `byte`. For example, `byte y = 5;` is perfectly valid. That's because the language specifies that for constant integer expressions (including integer literals) within the range of `byte`, there's an implicit conversion from the expression to `byte`. If you can get your head around that, you can apply the exact same idea to verbatim string literals.

When the compiler needs to convert an interpolated string literal into a *FormattableString*, it performs most of the same steps as for a conversion to *string*. But instead of *string.Format*, it calls the static *Create* method on the *System.Runtime.CompilerServices.FormattableStringFactory* class. This is another type introduced at the same time as *FormattableString*. To go back to an earlier example, say you have this source code:

```
int x = 10;
int y = 20;
FormattableString formattable = $"x={x}, y={y}";
```

That's handled by the compiler as if you'd written the following code instead (with the appropriate namespaces, of course):

```
int x = 10;
int y = 20;
FormattableString formattable = FormattableStringFactory.Create(
    "x={0}, y={1}", x, y);
```

FormattableString is an abstract class with members as shown in the following listing.

Listing 9.5 Members declared by *FormattableString*

```
public abstract class FormattableString : IFormattable
{
    protected FormattableString();
    public abstract object GetArgument(int index);
    public abstract object[] GetArguments();
    public static string Invariant(FormattableString formattable);
    string IFormattable.ToString
        (string ignored, IFormatProvider formatProvider);
    public override string ToString();
    public abstract string ToString(IFormatProvider formatProvider);
    public abstract int ArgumentCount { get; }
    public abstract string Format { get; }
}
```

Now that you know when and how *FormattableString* instances are built, let's see what you can do with them.

9.3.2 *Formatting a FormattableString in a specific culture*

By far, the most common use for *FormattableString* will be to perform the formatting in an explicitly specified culture instead of in the default culture for the thread. I expect that most uses will be for a single culture: the invariant culture. This is so common that it has its own static method: *Invariant*. Calling this is equivalent to passing *CultureInfo.InvariantCulture* into the *ToString(IFormatProvider)* method, which behaves exactly as you'd expect. But making *Invariant* a static method means it's simpler to call as a subtle corollary of the language details you just looked at in section 9.3.1. The fact that it takes *FormattableString* as a

parameter means you can just use an interpolated string literal as an argument, and the compiler knows that it has to apply the relevant conversion; there's no need for a cast or a separate variable.

Let's consider a concrete example to make it clear. Suppose you have a `DateTime` value and you want to format just the date part of it in ISO-8601 format as part of a URL query parameter for machine-to-machine communication. You want to use the invariant culture to avoid any unexpected results from using the default culture.

NOTE Even when you specify a custom format string for a date and time, and even when that custom format uses only digits, the culture still has an impact. The biggest one is that the value is represented in the default calendar system for the culture. If you format October 21, 2016 (Gregorian) in the culture `ar-SA` (Arabic in Saudi Arabia), you'll get a result with a year of 1438.

You can do this formatting in four ways, all of which are shown together in the following listing. All four approaches give exactly the same result, but I've shown all of them to demonstrate how the multiple language features work together to give a clean final option.

Listing 9.6 Formatting a date in the invariant culture

```
DateTime date = DateTime.UtcNow;

string parameter1 = string.Format(
    CultureInfo.InvariantCulture,
    "x={0:yyyy-MM-dd}",
    date);
```

**Old-school formatting
with `string.Format`**

```
string parameter2 =
    ((FormattableString)$"x={date:yyyy-MM-dd}")
    .ToString(CultureInfo.InvariantCulture);
```

**Casting to `FormattableString` and
calling `ToString(IFormatProvider)`**

```
string parameter3 = FormattableString.Invariant(
    $"x={date:yyyy-MM-dd}");
```

**Regular call to
`FormattableString.Invariant`**

```
string parameter4 = Invariant($"x={date:yyyy-MM-dd}");
```

**Shortened call to
`FormattableString.Invariant`**

The main interesting difference is between the initializers for `parameter2` and `parameter3`. To make sure you have a `FormattableString` for `parameter2` rather than just a `string`, you have to cast the interpolated string literal to that type. An alternative would've been to declare a separate local variable of type `FormattableString`, but that would've been about as long-winded. Compare that with the way `parameter3` is initialized, which uses the `Invariant` method that accepts a parameter of type `FormattableString`. That allows the compiler to infer that you want to use the implicit conversion from an interpolated string literal to `FormattableString`, because that's the only way that the call will be valid.

I've cheated for `parameter4`. I've used a feature you haven't seen yet, making static methods from a type available with a `using static` directive. You can flick

forward to the details later (section 10.1.1) or trust me for now that it works. You just need using `System.FormattableString` in your list of using directives.

FORMATTING IN A NONINVARIANT CULTURE

If you want to format a `FormattableString` in any culture other than the invariant one, you need to use one of the `ToString` methods. In most cases, you'll want to call the `ToString(IFormatProvider)` overload directly. As a slightly shorter example than you saw earlier, here's code to format the current date and time in US English using the "general date/time with short time" standard format string ("g"):

```
FormattableString fs = $"The current date and time is: {DateTime.Now:g}";  
string formatted = fs.ToString(CultureInfo.GetCultureInfo("en-US"));
```

Occasionally, you may want to pass the `FormattableString` to another piece of code to perform the final formatting step. In that case, it's worth remembering that `FormattableString` implements the `IFormattable` interface, so any method accepting an `IFormattable` will accept a `FormattableString`. The `FormattableString` implementation of `IFormattable.ToString(string, IFormatProvider)` ignores the string parameter because it already has everything it needs: it uses the `IFormatProvider` parameter to call the `ToString(IFormatProvider)` method.

Now that you know how to use cultures with interpolated string literals, you may be wondering why the other members of `FormattableString` exist. In the next section, you'll look at one example.

9.3.3 Other uses for `FormattableString`

I'm not expecting `FormattableString` to be widely used outside the culture scenario I showed in section 9.3.2, but it's worth considering what *can* be done. I've chosen this example as one that's immediately recognizable and elegant in its own way, but I wouldn't go so far as to recommend its use. Aside from the code presented here lacking validation and some features, it may give the wrong impression to a casual reader (and to static code analysis tools). By all means, pursue it as an idea, but use appropriate caution.

Most developers are aware of SQL injection attacks as a security vulnerability, and many know the common solution in the format of parameterized SQL. Listing 9.7 shows what you don't want to do. If a user enters a value containing an apostrophe, they have huge amounts of power over your database. Imagine that you have a database with entries of some kind that a user can add a tag to partitioned by user identifier. You're trying to list all the descriptions for a user-specified tag restricted to that user.

Listing 9.7 Awooga! Awooga! Do not use this code!

```
var tag = Console.ReadLine();  
using (var conn = new SqlConnection(connectionString)) {  
    conn.Open();  
}
```

↳ **Reads arbitrary data from the user**

```

string sql =
    $"@SELECT Description FROM Entries
        WHERE Tag='{tag}' AND UserId={userId}";
using (var command = new SqlCommand(sql, conn))
{
    using (var reader = command.ExecuteReader())
    {
        ...
    }
}

```

Most SQL injection vulnerabilities I've seen in C# use string concatenation rather than string formatting, but it's the same deal. It mixes code (SQL) and data (the value the user entered) in an alarming way.

I'm going to assume that you know how you'd have fixed this problem in the past using parameterized SQL and calling `command.Parameters.Add(...)` appropriately. Code and data are suitably separated, and life is good again. Unfortunately, that safe code doesn't look as appealing as the code in listing 9.7. What if you could have it both ways? What if you could write SQL that made it obvious what you were trying to do but was still safely parameterized? With `FormattableString`, you can do exactly that.

You'll work backward, from our desired user code, through the implementation that enables it. The following listing shows the soon-to-be-safe equivalent of listing 9.7.

Listing 9.8 Safe SQL parameterization using `FormattableString`

```

var tag = Console.ReadLine();
using (var conn = new SqlConnection(connectionString))
{
    conn.Open();
    using (var command = conn.NewSqlCommand(
        $"@SELECT Description FROM Entries
            WHERE Tag={tag:NVarChar}
            AND UserId={userId:Int}""))
    {
        using (var reader = command.ExecuteReader())
        {
            // Use the data
        }
    }
}

```

Most of this listing is identical to listing 9.7. The only difference is in how you construct the `SqlCommand`. Instead of using an interpolated string literal to format the values into SQL and then passing that string into the `SqlCommand` constructor, you're using a new method called `NewSqlCommand`, which is an extension method you'll write soon. Predictably, the second parameter of that method isn't `string` but `FormattableString`. The interpolated string literal no longer has apostrophes around `{tag}`, and you've specified each parameter's database type as a format string. That's certainly unusual. What is it doing?

First, let's think about what the compiler is doing for you. It's splitting the interpolated string literal into two parts: a composite format string and the arguments for the format items. The composite format string the compiler creates will look like this:

```
SELECT Description FROM Entries  
WHERE Tag={0:NVarChar} AND UserId={1:Int}
```

You want SQL that ends up looking like this instead:

```
SELECT Description FROM Entries  
WHERE Tag=@p0 AND UserId=@p1
```

That's easy enough to do; you just need to format the composite format string, passing in arguments that will evaluate to "@p0" and "@p1". If the type of those arguments implements *IFormattable*, calling *string.Format* will pass the *NVarChar* and *Int* format strings as well, so you can set the types of the *SqlParameter* objects appropriately. You can autogenerate the names, and the values come directly from the *FormattableString*.

It's highly unusual to make an *IFormattable.ToString* implementation have side effects, but you're using only this format-capturing type for this single call, and you can keep it safely hidden from any other code. The following listing is a complete implementation.

Listing 9.9 Implementing safe SQL formatting

```
public static class SqlFormattableString  
{  
    public static SqlCommand NewSqlCommand(  
        this SqlConnection conn, FormattableString formattableString)  
    {  
        SqlParameter[] sqlParameters = formattableString.GetArguments()  
            .Select((value, position) =>  
                new SqlParameterInvariant($"@p{position}", value))  
            .ToArray();  
        object[] formatArguments = sqlParameters  
            .Select(p => new FormatCapturingParameter(p))  
            .ToArray();  
        string sql = string.Format(formattableString.Format,  
            formatArguments);  
        var command = new SqlCommand(sql, conn);  
        command.Parameters.AddRange(sqlParameters);  
        return command;  
    }  
  
    private class FormatCapturingParameter : IFormattable  
    {  
        private readonly SqlParameter parameter;  
  
        internal FormatCapturingParameter(SqlParameter parameter)  
        {  
            this.parameter = parameter;  
        }  
    }  
}
```

```
        public string ToString(string format, IFormatProvider formatProvider)
        {
            if (!string.IsNullOrEmpty(format))
            {
                parameter.SqlDbType = (SqlDbType) Enum.Parse(
                    typeof(SqlDbType), format, true);
            }
            return parameter.ParameterName;
        }
    }
```

The only public part of this is the `SqlFormattableString` static class with its `NewSqlCommand` method. Everything else is a hidden implementation detail. For each placeholder in the format string, you create a `SqlParameter` and a corresponding `FormatCapturingParameter`. The latter is used to format the parameter name in the SQL as `@p0`, `@p1`, and so on, and the value provided to the `ToString` method is set into the `SqlParameter`. The type of the parameter is also set if the user specifies it in the format string.

At this point, you need to make up your own mind as to whether this is something you'd like to see in your production codebase. I'd want to implement extra features (such as including the size in the format string; you can't use the alignment part of a format item, because `string.Format` handles that itself), but it can certainly be productionized appropriately. But is it just too clever? Are you going to have to walk every new developer on the project through this, saying, "Yes, I know it looks like we have a massive SQL injection vulnerability, but it's okay, really"?

Regardless of this specific example, you may well be able to find similar situations for which you can use the compiler's extraction of the data and separation from the text of an interpolated string literal. Always think carefully about whether a solution like this is really providing a benefit or whether it's just giving you a chance to feel smart.

All of this is useful if you’re targeting .NET 4.6, but what if you’re stuck on an older framework version? Just because you’re using a C# 6 compiler doesn’t mean you’re necessarily targeting a modern version of the framework. Fortunately, the C# compiler doesn’t tie this to a specific framework version; it just needs the right types to be available *somewhere*.

9.3.4 Using FormattableString with older versions of .NET

Just like the attribute for extension methods and caller information attributes, the C# compiler doesn't have a fixed idea of which assembly should contain the `FormattableString` and `FormattableStringFactory` types it relies on. The compiler cares about the namespaces and expects an appropriate static `Create` method to be present on `FormattableStringFactory`, but that's about it. If you want to take advantage of the benefits of `FormattableString` but you're stuck targeting an earlier version of the framework, you can implement both types yourself.

Before I show you the code, I should point out that this should be viewed as a last resort. When you eventually upgrade your environment to target .NET 4.6, you should remove these types immediately to avoid compiler warnings. Although you can get away with having your own implementation even if you end up executing in .NET 4.6, I'd try to avoid getting into that situation; in my experience, having the same type in different assemblies can lead to issues that are hard to diagnose.

With all the caveats out of the way, the implementation is simple. Listing 9.10 shows both types. I haven't included any validation, I've made *FormattableString* a concrete type for brevity, and I've made both classes internal, but the compiler doesn't mind those changes. The reason for making the types internal is to avoid other assemblies taking a dependency on your implementation; whether that's suitable for your precise situation is hard to predict, but please consider it carefully before making the types public.

Listing 9.10 Implementing *FormattableString* from scratch

```
using System.Globalization;

namespace System.Runtime.CompilerServices
{
    internal static class FormattableStringFactory
    {
        internal static FormattableString Create(
            string format, params object[] arguments) =>
            new FormattableString(format, arguments);
    }
}

namespace System
{
    internal class FormattableString : IFormattable
    {
        public string Format { get; }
        private readonly object[] arguments;

        internal FormattableString(string format, object[] arguments)
        {
            Format = format;
            this.arguments = arguments;
        }

        public object GetArgument(int index) => arguments[index];
        public object[] GetArguments() => arguments;
        public int ArgumentCount => arguments.Length;
        public static string Invariant(FormattableString formattable) =>
            formattable?.ToString(CultureInfo.InvariantCulture);
        public string ToString(IFormatProvider formatProvider) =>
            string.Format(formatProvider, Format, arguments);
        public string ToString(
            string ignored, IFormatProvider formatProvider) =>
            ToString(formatProvider);
    }
}
```

I won't explain the details of the code, because each individual member is quite simple. The only part that may need a little explanation is in the `Invariant` method calling `formattable?.ToString(CultureInfo.InvariantCulture)`. The `?.` part of this expression is the *null conditional* operator, which you'll look at in more detail in section 10.3. Now you know everything you can do with interpolated string literals, but what about what you should do with them?

9.4 **Uses, guidelines, and limitations**

Like expression-bodied members, interpolated string literals are a safe feature to experiment with. You can adjust your code to meet your own (or team-wide) thresholds. If you change your mind later and want to go back to the old code, doing so is trivial. Unless you start using `FormattableString` in your APIs, the use of interpolated string literals is a hidden implementation detail. That doesn't mean it should be used absolutely everywhere, of course. In this section, we'll discuss where it makes sense to use interpolated string literals, where it doesn't, and where you might find you can't even if you want to.

9.4.1 **Developers and machines, but maybe not end users**

First, the good news: almost anywhere you're already using string formatting with hardcoded composite format strings or anywhere you're using plain string concatenation you can use interpolated strings. Most of the time, the code will be more readable afterward.

The hardcoded part is important here. Interpolated string literals aren't dynamic. The composite format string is there inside your source code; the compiler just mangles it a little to use regular format items. That's fine when you know the text and format of the desired string beforehand, but it's not flexible.

One way of categorizing strings is to think about who or what is going to consume them. For the purposes of this section, I'll consider three consumers:

- Strings designed for other code to parse
- Messages for other developers
- Messages for end users

Let's look at each kind of string in turn and think about whether interpolated string literals are useful.

MACHINE-READABLE STRINGS

Lots of code is built to read other strings. There are machine-readable log formats, URL query parameters, and text-based data formats such as XML, JSON, or YAML. All of these have a set format, and any values should be formatted using the invariant culture. This is a great place to use `FormattableString`, as you've already seen, if you need to perform the formatting yourself. As a reminder, you should typically be taking advantage of an appropriate API for the formatting of machine-readable strings anyway.

Bear in mind that each of these strings might also contain nested strings aimed at humans; each line of a log file may be formatted in a specific way to make it easy to

treat as a single record, but the message part of it may be aimed at other developers. You need to keep track of what level of nesting each part of your code is working at.

MESSAGES FOR OTHER DEVELOPERS

If you look at a large codebase, you’re likely to find that many of your string literals are aimed at other developers, whether they’re colleagues within the same company or developers using an API you’ve released. These are primarily as follows:

- Tooling strings such as help messages in console applications
- Diagnostic or progress messages written to logs or the console
- Exception messages

In my experience, these are typically in English. Although some companies—including Microsoft—go to the trouble of localizing their error messages, most don’t. Localization has a significant cost both in terms of the data translation and the code to use it properly. If you know your audience is at least reasonably comfortable reading English, and particularly if they may want to share the messages on English-oriented sites such as Stack Overflow, it’s usually not worth the effort of localizing these strings.

Whether you go so far as making sure that the values within the text are all formatted in a fixed culture is a different matter. It can definitely help to improve consistency, but I suspect I’m not the only developer who doesn’t pay as much attention to that as I might. I encourage you to use a nonambiguous format for dates, however. The ISO format of yyyy-MM-dd is easy to understand and doesn’t have the “month first or day first?” problem of dd/MM/yyyy or MM/dd/yyyy. As I noted earlier, the culture can affect which numbers are produced because of different calendar systems being in use in different parts of the world. Consider carefully whether you want to use the invariant culture to force the use of the Gregorian calendar. For example, code to throw an exception for an invalid argument might look like this:

```
throw new ArgumentExceptionInvariant(  
    $"Start date {start:yyyy-MM-dd} should not be earlier than year 2000."))
```

If you know that all the developers reading these strings are going to be in the same non-English culture, it’s entirely reasonable to write all those messages in that culture instead.

MESSAGES FOR END USERS

Finally, almost all applications have at least some text that’s displayed to an end user. As with developers, you need to be aware of the expectations of each user in order to make the right decision for how to present text to them. In some cases, you can be confident that all your end users are happy to use a single culture. This is typically the case if you’re building an application to be used internally within a business or other organization that’s based in one location. Here it’s much more likely that you’ll use whatever that local culture is rather than English, but you don’t need to worry about two users wanting to see the same information presented in different ways.

So far, all these situations have been amenable to interpolated string literals. I’m particularly fond of using them for exception messages. They let me write concise

code that still provides useful context to the unfortunate developer poring over logs and trying to work out what's gone wrong this time.

But interpolated string literals are rarely helpful when you have end users in multiple cultures, and they can hurt your product if you don't localize. Here, the format strings are likely to be in resource files rather than in your code anyway, so you're unlikely to even see the possibility of using interpolated string literals. There are occasional exceptions to this, such as when you're formatting just one snippet of information to put within a specific HTML tag or something similar. In those exceptional cases, an interpolated string literal should be fine, but don't expect to use them much.

You've seen that you can't use interpolated string literals for resource files. Next, you'll look at other cases for which the feature simply isn't designed to help you.

9.4.2 Hard limitations of interpolated string literals

Every feature has its limits, and interpolated string literals are no exception. These limitations sometimes have workarounds, which I'll show you before generally advising you not to try them in the first place.

NO DYNAMIC FORMATTING

You've already seen that you can't change most of the composite format string that makes up the interpolated string literal. Yet one piece feels like it should be expressible dynamically but isn't: individual format strings. Let's take one piece of an example from earlier:

```
Console.WriteLine($"Price: {price,9:C}");
```

Here, I've chosen 9 as the alignment, knowing that the values I'd be formatting would fit nicely into nine characters. But what if you know that sometimes all the values you need to format will be small and other times they may be huge? It'd be nice to make that 9 part dynamic, but there's no simple way of doing it. The closest you can easily come is to use an interpolated string literal as the input to `string.Format` or the equivalent `Console.WriteLine` overload, as in the following example:

```
int alignment = GetAlignmentFromValues(allTheValues);
Console.WriteLine($"Price: {{0,{alignment}:C}}", price);
```

The first and last braces are doubled as the escape mechanism in string formats, because you want the result of the interpolated string literal to be a string such as `"Price: {0,9}"` that's ready to be formatted using the `price` variable to fill in the format item. This isn't code I'd want to either write or read.

NO EXPRESSION REEVALUATION

The compiler always converts an interpolated string literal into code that immediately evaluates the expressions in the format items and uses them to build either a `string` or a `FormattableString`. The evaluation can't be deferred or repeated. Consider

the short example in the following listing. It prints the same value twice, even though the developer may expect it to use deferred execution.

Listing 9.11 Even `FormattableString` evaluates expressions eagerly

```
string value = "Before";
FormattableString formattable = $"Current value: {value}";
Console.WriteLine(formattable);           ← Prints "Current
                                         value: Before"
value = "After";
Console.WriteLine(formattable);           ← Still prints "Current
                                         value: Before"
```

If you're desperate, you can work around this. If you change the expression to include a lambda expression that captures `value`, you can abuse this to evaluate it each time it's formatted. Although the lambda expression itself is converted into a delegate immediately, the resulting delegate would capture the `value` variable, not its current value, and you can force the delegate to be evaluated each time you format the `FormattableString`. This is a sufficiently bad idea that, although I've included an example of it in the downloadable samples for the book, I'm not going to sully these pages with it. (It's still a fun abuse, admittedly.)

NO BARE COLONS

Although you can use pretty much any expression computing a value in interpolated string literals, there's one problem with the conditional `? :` operator: it confuses the compiler and indeed the grammar of the C# language. Unless you're careful, the colon ends up being handled as the separator between the expression and the format string, which leads to a compile-time error. For example, this is invalid:

```
Console.WriteLine($"Adult? {age >= 18 ? "Yes" : "No"}");
```

It's simple to fix by using parentheses around the conditional expression:

```
Console.WriteLine($"Adult? {(age >= 18 ? "Yes" : "No")});
```

I rarely find this to be a problem, partly because I usually try to keep the expressions shorter than this anyway. I'd probably extract the yes/no value into a separate string variable first. This leads us nicely into a discussion about when the choice of whether to use an interpolated string literal really comes down to a matter of taste.

9.4.3 When you can but really shouldn't

The compiler isn't going to mind if you abuse interpolated string literals, but your coworkers might. There are two primary reasons not to use them even where you can.

DEFER FORMATTING FOR STRINGS THAT MAY NOT BE USED

Sometimes you want to pass a format string and the arguments that would be formatted to a method that might use them or might not. For example, if you have a precondition

validation method, you might want to pass in the condition to check along with the format and arguments of an exception message to create if (and only if) the condition fails. It's easy to write code like this:

```
Preconditions.CheckArgument(
    start.Year < 2000,
    Invariant($"Start date {start:yyyy-MM-dd} should not be earlier than year
    ➔ 2000."));
```

Alternatively, you could have a logging framework that'll log only if the level has been configured appropriately at execution time. For example, you might want to log the size of a request that your server has received:

```
Logger.Debug("Received request with {0} bytes", request.Length);
```

You might be tempted to use an interpolated string literal for this by changing the code to the following:

```
Logger.Debug($"Received request with {request.Length} bytes");
```

That would be a bad idea; it forces the string to be formatted even if it's just going to be thrown away, because the formatting will unconditionally be performed before the method is called rather than within the method only if it's needed. Although string formatting isn't hugely expensive in terms of performance, you don't want to be doing it unnecessarily.

You may be wondering whether `FormattableString` would help here. If the validation or logging library accepted a `FormattableString` as an input parameter, you could defer the formatting and control the culture used for formatting in a single place. Although that's true, it'd still involve creating the object each time, which is still an unnecessary cost.

FORMAT FOR READABILITY

The second reason for not using interpolated string literals is that they can make the code harder to read. Short expressions are absolutely fine and help readability. But when the expression becomes longer, working out which parts of the literal are code and which are text starts to take more time. I find that parentheses are the killer; if you have more than a couple of method or constructor calls in the expression, they end up being confusing. This goes double when the text also includes parentheses.

Here's a real example from Noda Time. It's in a test rather than in production code, but I still want the tests to be readable:

```
private static string FormatMemberDebugName(MemberInfo m) =>
    string.Format("{0}.{1}({2})",
        m.DeclaringType.Name,
        m.Name,
        string.Join(", ", GetParameters(m).Select(p => p.ParameterType)));
```

That's not too bad, but imagine putting the three arguments within the string. I've done it, and it's not pretty; you end up with a literal that's more than 100 characters

long. You can't break it up to use vertical formatting to make each argument stand alone as I have with the preceding layout, so it ends up being noise.

To give one final tongue-in-cheek example of just how bad an idea it can be, remember the code used to start the chapter:

```
Console.WriteLine("What's your name? ");
string name = Console.ReadLine();
Console.WriteLine("Hello, {0}!", name);
```

You can put all of this inside a single statement using an interpolated string literal. You may be skeptical; after all, that code consists of three separate statements, and the interpolated string literal can include only expressions. That's true, but statement-bodied lambda expressions are still expressions. You need to cast the lambda expression to a specific delegate type, and then you need to invoke it to get the result, but it's all doable. It's just not pleasant. Here's one option, which does at least use separate lines for each statement by virtue of a verbatim interpolated string literal, but that's about all that can be said in its favor:

```
Console.WriteLine($@"Hello {((Func<string>) () =>
{
    Console.Write("What's your name? ");
    return Console.ReadLine();
}))()!" );
```

I thoroughly recommend running: run the code to prove that it works, and then run away from it as fast as you can. While you're recovering from that, let's look at the other string-oriented feature of C# 6.

9.5 Accessing identifiers with nameof

The `nameof` operator is trivial to describe: it takes an expression referring to a member or local variable, and the result is a compile-time constant string with the simple name for that member or variable. It's that simple. Anytime you hardcode the name of a class, property, or method, you'll be better off with the `nameof` operator. Your code will be more robust both now and in the face of changes.

9.5.1 First examples of nameof

In terms of syntax, the `nameof` operator is like the `typeof` operator, except that the identifier in the parentheses doesn't have to be a type. The following listing shows a short example with a few kinds of members.

Listing 9.12 Printing out the names of a class, method, field, and parameter

```
using System;

class SimpleNameof
{
    private string field;
```

```

static void Main(string[] args)
{
    Console.WriteLine(nameof(SimpleNameof));
    Console.WriteLine(nameof(Main));
    Console.WriteLine(nameof(args));
    Console.WriteLine(nameof(field));
}
}

```

The result is exactly what you'd probably expect:

```

SimpleNameof
Main
args
field

```

So far, so good. But, obviously, you could've achieved the same result by using string literals. The code would've been shorter, too. So why is it better to use `nameof`? In one word, robustness. If you make a typo in a string literal, there's nothing to tell you, whereas if you make a typo in a `nameof` operand, you'll get a compile-time error.

NOTE The compiler still won't be able to spot the problem if you refer to a different member with a similar name. If you have two members that differ only in case, such as `filename` and `fileName`, you can easily refer to the wrong one without the compiler noticing. This is a good reason to avoid such similar names, but it's always been a bad idea to name things so similarly; even if you don't confuse the compiler, you can easily confuse a human reader.

Not only will the compiler tell you if you get things wrong, but it knows that your `nameof` code is associated with the member or variable you're naming. If you rename it in a refactoring-aware way, your `nameof` operand will change, too.

For example, consider the following listing. Its purpose is irrelevant, but note that `oldName` occurs three times: for the parameter declaration, obtaining its name with `nameof`, and obtaining the value as a simple expression.

Listing 9.13 A simple method using its parameter twice in the body

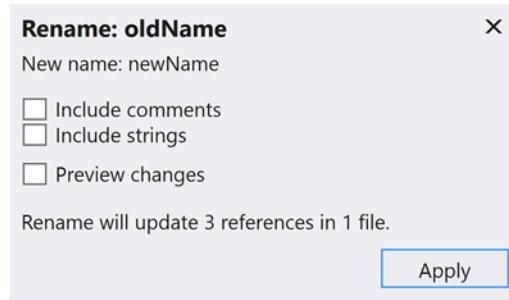
```

static void RenameDemo(string oldName)
{
    Console.WriteLine($"nameof({oldName}) = {oldName}");
}

```

In Visual Studio, if you place your cursor within any of the three occurrences of `oldName` and press F2 for the Rename operation, all three will be renamed together, as shown in figure 9.2.

The same approach works for other names (methods, types, and so forth). Basically, `nameof` is refactoring friendly in a way that hardcoded string literals aren't. But when should you use it?



```
static void RenameDemo(string newName)
{
    Console.WriteLine($"{nameof(newName)} = {newName}");
}
```

Figure 9.2 Renaming an identifier in Visual Studio

9.5.2 Common uses of nameof

I'm not going to claim that the examples here are the only sensible uses of `nameof`. They're just the ones I've come across most often. They're mostly places where prior to C# 6, you'd have seen either hardcoded names or, possibly, expression trees being used as a workaround that's refactoring friendly but complex.

ARGUMENT VALIDATION

In chapter 8, when I showed the uses of `Preconditions.CheckNotNull` in Noda Time, that wasn't the code that's actually in the library. The real code includes the name of the parameter with the null value, which makes it a lot more useful. The `InZone` method I showed there looks like this:

```
public ZonedDateTime InZone(
    DateTimeZone zone,
    ZoneLocalMappingResolver resolver)
{
    Preconditions.CheckNotNull(zone, nameof(zone));
    Preconditions.CheckNotNull(resolver, nameof(resolver));
    return zone.ResolveLocal(this, resolver);
}
```

Other precondition methods are used in a similar way. This is by far the most common use I find for `nameof`. If you're not already validating arguments to your public methods, I strongly advise you to start doing so; `nameof` makes it easier than ever to perform robust validation with informational messages.

PROPERTY CHANGE NOTIFICATION FOR COMPUTED PROPERTIES

As you saw in section 7.2, `CallerMemberNameAttribute` makes it easy to raise events in `INotifyPropertyChanged` implementations when the property itself

changes. But what if changing the value of one property has an effect on another property? For example, suppose you have a `Rectangle` class with read/write `Height` and `Width` properties and a read-only `Area` property. It's useful to be able to raise the event for the `Area` property and specify the name of the property in a safe way, as shown in the following listing.

Listing 9.14 Using `nameof` to raise a property change notification

```
public class Rectangle : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    private double width;
    private double height;

    public double Width
    {
        get { return width; }
        set
        {
            if (width == value)           ← Avoid raising events when
                                         the value isn't changing.
            {
                return;
            }
            width = value;
            RaisePropertyChanged();      ← Raises the event
                                         for the Width
                                         property
            RaisePropertyChanged(nameof(Area)); ← Raises the event
                                         for the Area
                                         property
        }
    }

    public double Height { ... }   ← Implemented just
                                 like Width

    public double Area => Width * Height;   ← Computed
                                              property

    private void RaisePropertyChanged(
        [CallerMemberName] string propertyName = null) { ... }

}
```

**Change notification
as per section 7.2**

Most of this listing is exactly as you'd have written it in C# 5, but the line in bold would've had to be `RaisePropertyChanged("Area")` or `RaisePropertyChanged(() => Area)`. The latter approach would've been complex, in terms of the `RaisePropertyChanged` code, and inefficient, because it builds up an expression tree solely to be inspected for the name. The `nameof` solution is much cleaner.

ATTRIBUTES

Sometimes attributes refer to other members to indicate how the members relate to each other. When you want to refer to a type, you can already use `typeof` to make that relationship, but that doesn't work for any other kind of member. As a concrete

example, NUnit allows tests to be parameterized with values that are extracted from a field, property, or method using the `TestCaseSource` attribute. The `nameof` operator allows you to refer to that member in a safe way. The following listing shows yet another example from Noda Time, testing that all the time zones loaded from the Time Zone Database (TZDB, now hosted by IANA) behave appropriately at the start and end of time.

Listing 9.15 Specifying a test case source with `nameof`

```
static readonly IEnumerable<DateTimeZone> AllZones =           | Field to retrieve all
    DateTimeZoneProviders.Tzdb.GetAllZones();                   | TZDB time zones

[TestMethod]                                         | Refers to the field
[TestCaseSource(nameof(AllZones))]      ←             | using nameof
public void AllZonesStartAndEnd(DateTimeZone zone)   ←             | Test method called with
{                                                 | each time zone in turn
    ...
}                                              ←             | Body of test
                                                | method omitted
```

The utility here isn't restricted to testing. It's applicable wherever attributes indicate a relationship. You could imagine a more sophisticated `RaisePropertyChanged` method from the preceding section, where the relationship between properties could be specified with attributes instead of within code:

```
[DerivedProperty(nameof(Area))
public double Width { ... }
```

The event-raising method could keep a cached data structure indicating that whenever it was notified that the `Width` property had changed, it should raise a change notification for `Area` as well.

Similarly, in object-relational mapping technologies such as Entity Framework, it's reasonably common to have two properties in a class: one for a foreign key and the other to be the entity that key represents. This is shown in the following example:

```
public class Employee
{
    [ForeignKey(nameof(Employer))]
    public Guid EmployerId { get; set; }
    public Company Employer { get; set; }
}
```

There are no doubt many other attributes that can take advantage of this approach. Now that you're aware of it, you may find places in your existing codebase that'll benefit from `nameof`. In particular, you should look for code where you need to use reflection with names that you do know at compile time but haven't previously been able to specify in a clean way. There are still a few little subtleties to cover for the sake of completeness, however.

9.5.3 Tricks and traps when using nameof

You may never need to know any of the details in this section. This content is primarily here just in case you find yourself surprised by the behavior of `nameof`. In general, it's a pretty simple feature, but a few aspects might surprise you.

REFERRING TO MEMBERS OF OTHER TYPES

Often, it's useful to be able to refer to members in one type from within code in another type. Going back to the `TestCaseSource` attribute, for example, in addition to a name, you can specify a type where NUnit will look for that name. If you have a source of information that'll be used from multiple tests, it makes sense to put it in a common place. To do this with `nameof`, you qualify it with the type as well. The result will be the simple name:

```
[TestCaseSource(typeof(Cultures), nameof(Cultures.AllCultures))]
```

That is equivalent to the following, except for all the normal benefits of `nameof`:

```
[TestCaseSource(typeof(Cultures), "AllCultures")]
```

You can also use a variable of the relevant type to access a member name, although only for instance members. In reverse, you can use the name of the type for both static and instance members. The following listing shows all the valid permutations.

Listing 9.16 All the valid ways of accessing names of members in other types

```
class OtherClass
{
    public static int StaticMember => 3;
    public int InstanceMember => 3;
}

class Qualifiednameof
{
    static void Main()
    {
        OtherClass instance = null;
        Console.WriteLine(nameof(instance.InstanceMember));
        Console.WriteLine(nameof(OtherClass.StaticMember));
        Console.WriteLine(nameof(OtherClass.InstanceMember));
    }
}
```

I prefer to always use the type name where possible; if you use a variable instead, it *looks* like the value of the variable may matter, but really it's used only at compile time to determine the type. If you're using an anonymous type, there's no type name you could use, so you have to use the variable.

A member still has to be accessible for you to refer to it using `nameof`; if `StaticMember` or `InstanceMember` in listing 9.16 had been private, the code trying to access their names would've failed to compile.

GENERICs

You may be wondering what happens if you try to take the name of a generic type or method and how it has to be specified. In particular, `typeof` allows both bound and unbound type names to be used; `typeof(List<string>)` and `typeof(List<>)` are both valid and give different results.

With `nameof`, the type argument must be specified but isn't included in the result. Additionally, there's no indication of the number of type parameters in the result: `nameof(Action<string>)` and `nameof(Action<string, string>)` both have a value of just "Action". This can be irritating, but it removes any question of how the resulting name should represent arrays, anonymous types, further generic types, and so on.

It seems likely to me that the requirement for a type argument to be specified may be removed in the future both to be consistent with `typeof` and to avoid having to specify a type that makes no difference to the result. But changing the result to include the number of type arguments or the type arguments themselves would be a breaking change, and I don't envision that happening. In most cases where that matters, using `typeof` to obtain a `Type` would be preferable anyway.

You can use a type parameter with a `nameof` operator, but unlike `typeof(T)`, it'll always return the name of the type parameter rather than the name of the type argument used for that type parameter at execution time. Here's a minimal example of that:

```
static string Method<T>() => nameof(T);
```



Always
returns "T"

It doesn't matter how you call the method: `Method<Guid>()` or `Method<Button>()` will both return "T".

USING ALIASES

Usually, using directives providing type or namespace aliases have no effect at execution time. They're just different ways of referring to the same type or namespace. The `nameof` operator is one exception to this rule. The output of the following listing is `GuidAlias`, not `Guid`.

Listing 9.17 Using an alias in the `nameof` operator

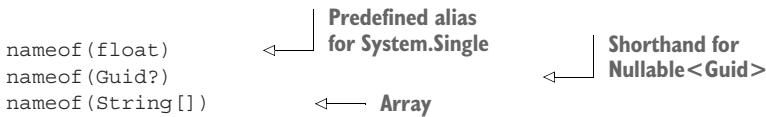
```
using System;

using GuidAlias = System.Guid;

class Test
{
    static void Main()
    {
        Console.WriteLine(nameof(GuidAlias));
    }
}
```

PREDEFINED ALIASES, ARRAYS AND NULLABLE VALUE TYPES

The `nameof` operator can't be used with any of the predefined aliases (`int`, `char`, `long`, and so on) or the `?` suffix to indicate a nullable value type or array types. Therefore, all the following are invalid:



These are a little annoying, but you have to use the CLR type name for the predefined aliases and the `Nullable<T>` syntax for nullable value types:

```
nameof(Single)
nameof(Nullable<Guid>)
```

As noted in the previous section on generics, the name of `Nullable<T>` will always be `Nullable` anyway.

THE NAME, THE SIMPLE NAME, AND ONLY THE NAME

The `nameof` operator is in some ways a cousin of the mythical `infoof` operator, which has never been seen outside the room used for C# language design meetings. (See <http://mng.bz/6GVe> for more information on `infoof`.) If the team ever manages to catch and tame `infoof`, it could return references to `MethodInfo`, `EventInfo`, `PropertyInfo` objects, and their friends. Alas, `infoof` has proved elusive so far, but many of the tricks it uses to evade capture aren't available to the simpler `nameof` operator. Trying to take the name of an overloaded method? That's fine; they all have the same name anyway. Can't easily resolve whether you're referring to a property or a type? Again, if they both have the same name, it doesn't matter which you use. Although `infoof` would certainly provide benefits above and beyond `nameof` if it could ever be sensibly designed, the `nameof` operator is considerably simpler and still addresses many of the same use cases.

One point to note about what's returned—the *simple name* or “bit at the end” in less specification-like terminology: it doesn't matter if you use `nameof(Guid)` or use `nameof(System.Guid)` from within a class importing the `System` namespace. The result will still be only “`Guid`”.

NAMESPACES

I haven't given details about all the members that `nameof` can be used with, because it's the set you'd expect: basically, all members except finalizers and constructors. But because we normally think about members in terms of types and members within types, you may be surprised that you can take the name of a namespace. Yes, namespaces are also members—of other namespaces.

But given the preceding rule about only the simple name being returned, that isn't terribly useful. If you use `nameof(System.Collections.Generic)`, I suspect you want the result to be `System.Collections.Generic`, but in reality, it's just

Generic. I've never come across a type where this is useful behavior, but then it's rarely important to know a namespace as a compile-time constant anyway.

Summary

- Interpolated string literals allow you to write simpler string-formatting code.
- You can still use format strings in interpolated string literals to provide more formatting details, but the format string has to be known at compile time.
- Interpolated verbatim string literals provide a mixture of the features of interpolated string literals and verbatim string literals.
- The `FormattableString` type provides access to all the information required to format a string before the formatting takes place.
- `FormattableString` is usable out of the box in .NET 4.6 and .NET Standard 1.3, but the compiler will use it if you provide your own implementation in earlier versions.
- The `nameof` operator provides refactoring-friendly and typo-safe access to names within your C# code.

10

A smörgåsbord of features for concise code

This chapter covers

- Avoiding code clutter when referring to static members
- Being more selective in importing extension methods
- Using extension methods in collection initializers
- Using indexers in object initializers
- Writing far fewer explicit null checks
- Catching only exceptions you're really interested in

This chapter is a grab bag of features. No particular theme runs through it besides expressing your code's intention in ever leaner ways. The features in this chapter are the ones left over when all the obvious ways of grouping features have been used. That doesn't in any way undermine their usefulness, however.

10.1 Using static directives

The first feature we'll look at provides a simpler way of referring to static members of a type, including extension methods.

10.1.1 Importing static members

The canonical example for this feature is `System.Math`, which is a static class and so has only static members. You're going to write a method that converts from polar coordinates (an angle and a distance) to Cartesian coordinates (the familiar (x, y) model) using the more human-friendly degrees instead of radians to express the angle. Figure 10.1 gives a concrete example of how a single point is represented in both coordinate systems. Don't worry if you're not totally comfortable with the math part of this; it's just an example that uses a lot of static members in a short piece of code.

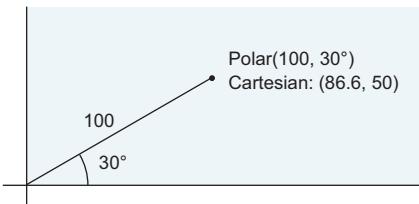


Figure 10.1 An example of polar and Cartesian coordinates

Assume that you already have a `Point` type representing Cartesian coordinates in a simple way. The conversion itself is fairly simple trigonometry:

- Convert the angle from degrees into radians by multiplying it by $\pi/180$. The constant π is available via `Math.PI`.
- Use the `Math.Cos` and `Math.Sin` methods to work out the x and y components of a point with magnitude 1, and multiply up.

The following listing shows the complete method with the uses of `System.Math` in bold. I've omitted the class declaration for convenience. It could be in a `CoordinateConverter` class, or it could be a factory method in the `Point` type itself.

Listing 10.1 Polar-to-Cartesian conversion in C# 5

```
using System;
...
static Point PolarToCartesian(double degrees, double magnitude)
{
    double radians = degrees * Math.PI / 180;
    return new Point(
        Math.Cos(radians) * magnitude,
        Math.Sin(radians) * magnitude);
}
```

Converts degrees
into radians

Trigonometry to
complete conversion

Although this code isn't terribly hard to read, you can imagine that as you write more math-related code, the repetition of `Math.` clutters the code considerably.

C# 6 introduced the *using static directive* to make this sort of code simpler. The following listing is equivalent to listing 10.1 but imports all the static members of `System.Math`.

Listing 10.2 Polar-to-Cartesian conversion in C# 6

```
using static System.Math;
...
static Point PolarToCartesian(double degrees, double magnitude)
{
    double radians = degrees * PI / 180;           ← Converts degrees
                                                    into radians
    return new Point(                                ← Trigonometry to
        Cos(radians) * magnitude,                  complete conversion
        Sin(radians) * magnitude);
}
```

As you can see, the syntax for a `using static` directive is simple:

```
using static type-name-or-alias;
```

With that in place, all the following members are available directly by using their simple names rather than having to qualify them with the type:

- Static fields and properties
- Static methods
- Enum values
- Nested types

The ability to use enum values directly is particularly useful in `switch` statements and anywhere you combine enum values. The following side-by-side example shows how to retrieve all the fields of a type with reflection. The text in bold highlights the code that can be removed with an appropriate `using static` directive.

C# 5 code	With <code>using static</code> in C# 6
<pre>using System.Reflection; ... var fields = type.GetFields(BindingFlags.Instance BindingFlags.Static BindingFlags.Public BindingFlags.NonPublic)</pre>	<pre>using static System.Reflection.BindingFlags; ... var fields = type.GetFields(Instance Static Public NonPublic);</pre>

Similarly, a `switch` statement responding to specific HTTP status codes can be made simpler by avoiding the repetition of the enum type name in every case label:

C# 5 code	With <code>using static</code> in C# 6
<pre>using System.Net; ... switch (response.StatusCode) { case HttpStatusCode.OK: ... case HttpStatusCode.TemporaryRedirect: case HttpStatusCode.Redirect: case HttpStatusCode.RedirectMethod: ... case HttpStatusCode.NotFound: ... default: ... }</pre>	<pre>using static System.Net.HttpStatusCode; ... switch (response.StatusCode) { case OK: ... case TemporaryRedirect: case Redirect: case RedirectMethod: ... case NotFound: ... default: ... }</pre>

Nested types are relatively rare in handwritten code, but they're more common in generated code. If you use them even occasionally, the ability to import them directly in C# 6 can significantly declutter your code. As an example, my implementation of the Google Protocol Buffers serialization framework to C# generates nested types to represent nested messages declared in the original .proto file. One quirk is that the nested C# types are doubly nested to avoid naming collisions. Say you have an original .proto file with a message like this:

```
message Outer {
    message Inner {
        string text = 1;
    }

    Inner inner = 1;
}
```

The code that's generated has the following structure with a lot more other members, of course:

```
public class Outer
{
    public static class Types
    {
        public class Inner
        {
            public string Text { get; set; }
        }
    }

    public Types.Inner Inner { get; set; }
}
```

To refer to `Inner` from your code in C# 5, you had to use `Outer.Types.Inner`, which is painful. The double nesting became considerably less inconvenient with C# 6, where it becomes relegated to a single `using static` directive:

```
using static Outer.Types;
...
Outer outer = new Outer { Inner = new Inner { Text = "Some text here" } };
```

In all these cases, the members that are available via the static imports are considered during member lookup only after other members have been considered. For example, if you have a static import of `System.Math` but you also have a `Sin` method declared in your class, a call to `Sin()` will find your `Sin` method rather than the one in `Math`.

The imported type doesn't have to be static

The `static` part of `using static` doesn't mean that the type you import must be static. The examples shown so far have been, but you can import regular types, too. That lets you access the static members of those types without qualification:

```
using static System.String;
...
string[] elements = { "a", "b" };
Console.WriteLine(Join(" ", elements));
```

Access `String.Join` by its simple name

I haven't found this to be as useful as the earlier examples, but it's available if you want it. Any nested types are made available by their simple names, too. There's one exception to the set of static members that's imported with a `using static` directive that isn't quite so straightforward, and that's extension methods.

10.1.2 Extension methods and using static

One aspect of C# 3 that I was never keen on was the way extension methods were discovered. Importing a namespace and importing extension methods were both performed with a single `using` directive; there was no way of doing one without the other and no way of importing extension methods from a single type. C# 6 improves the situation, although some of the aspects I dislike couldn't be fixed without breaking backward compatibility.

The two important ways in which extension methods and `using static` directives interact in C# 6 are easy to state but have subtle implications:

- Extension methods from a single type can be imported with a `using static` directive for that type without importing any extension methods from the rest of the namespace.
- Extension methods imported from a type aren't available as if you were calling a regular static method like `Math.Sin`. Instead, you have to call them as if they were instance methods on the extended type.

I'll demonstrate the first point by using the most commonly used set of extension methods in all of .NET: the ones for LINQ. The `System.Linq.Queryable` class contains extension methods for `IQueryable<T>` accepting expression trees, and the `System.Linq.Enumerable` class contains extension methods for `IEnumerable<T>` accepting delegates. Because `IQueryable<T>` inherits from `IEnumerable<T>` with a regular `using` directive for `System.Linq`, you can use the extension methods accepting delegates on `IQueryable<T>`, although you usually don't want to. The following listing shows how a `using static` directive for just `System.Linq.Queryable` means the extension methods in `System.Linq.Enumerable` aren't picked up.

Listing 10.3 Selective importing of extension methods

```
using static System.Linq.Queryable;
...
var query = new[] { "a", "bc", "d" }.AsQueryable();           ← Creates an
                                                               IQueryable<string>

Expression<Func<string, bool>> expr =
    x => x.Length > 1;                                     ← Creates a delegate
Func<string, bool> del = x => x.Length > 1;                and expression tree

var valid = query.Where(expr);
var invalid = query.Where(del);                            ← Invalid: no in-scope
                                                               Where method
                                                               accepts a delegate
                                                               ↑
                                                               Valid: uses
                                                               Queryable.Where
```

One point that's worth noting is that if you accidentally imported `System.Linq` with a regular `using` directive, such as to allow `query` to be explicitly typed, that would silently make the last line valid.

The impact of this change should be considered carefully by library authors. If you wish to include some extension methods but allow users to explicitly opt into them, I encourage the use of a separate namespace for that purpose. The good news is that you can now be confident that any users—at least those with C# 6—can be selective in which extension methods to import without you having to create many namespaces. For example, in Noda Time 2.0, I introduced a `NodaTime.Extensions` namespace with extension methods targeting many types. I expect that some users will want to import only a subset of those extension methods, so I split the method declarations into several classes, with each class containing methods extending a single type. In other cases, you may wish to split your extension methods along different lines. The important point is that you should consider your options carefully.

The fact that extension methods can't be called as if they were regular static methods is also easily demonstrated using LINQ. Listing 10.4 shows this by calling the `Enumerable.Count` method on a sequence of strings: once in a valid way as an extension method, as if it were an instance method declared in `IEnumerable<T>`, and once attempting to use it as a regular static method.

Listing 10.4 Attempting to call `Enumerable.Count` in two ways

```
using System.Collections.Generic;
using static System.Linq.Enumerable;
...
IEnumerable<string> strings = new[] { "a", "b", "c" };
int valid = strings.Count();                                ← Valid: calling Count as if it
int invalid = Count(strings);                            | were an instance method
                                                        | Invalid: extension methods aren't
                                                        | imported as regular static methods
```

Effectively, the language is encouraging you to think of extension methods as different from other static methods in a way that it didn't before. Again, this has an impact on library developers: converting a method that already existed in a static class into an extension method (by adding the `this` modifier to the first parameter) used to be a nonbreaking change. As of C# 6, that becomes a breaking change: callers who were importing the method with a `using static` directive would find that their code no longer compiled after the method became an extension method.

NOTE Extension methods discovered via static imports aren't preferred over extension methods discovered through namespace imports. If you make a method call that isn't handled by regular method invocation, but multiple extension methods are applicable via imported namespaces or classes, overload resolution is applied as normal.

Just like extension methods, object and collection initializers were largely added to the language as part of the bigger feature of LINQ. And just like extension methods, they've been tweaked in C# 6 to make them slightly more powerful.

10.2 Object and collection initializer enhancements

As a reminder, object and collection initializers were introduced in C# 3. Object initializers are used to set properties (or, more rarely, fields) in newly created objects; collection initializers are used to add elements to newly created collections via the `Add` methods that the collection type supports. The following simple example shows initializing a Windows Forms `Button` with text and a background color as well as initializing a `List<int>` with three values:

```
Button button = new Button { Text = "Go", BackColor = Color.Red };
List<int> numbers = new List<int> { 5, 10, 20 };
```

C# 6 enhances both of these features and makes them slightly more flexible. These enhancements aren't as globally useful as some of the other features in C# 6, but they're still welcome additions. In both cases, the initializers have been expanded to include members that previously couldn't be used there: object initializers can now use indexers, and collection initializers can now use extension methods.

10.2.1 Indexers in object initializers

Until C# 6, object initializers could invoke only property setters or set fields directly. C# 6 allows indexer setters to be invoked as well using the `[index] = value` syntax used to invoke them in regular code.

To demonstrate this in a simple way, I'll use `StringBuilder`. This would be a fairly unusual usage, but we'll talk about best practices shortly. The example initializes a `StringBuilder` from an existing string ("This text needs truncating"), truncates the builder to a set length, and modifies the last character to a Unicode ellipsis (...). When printed to the console, the result is "This text...". Before C# 6, you couldn't have modified the last character within the initializer, so you would've ended up with something like this:

```
string text = "This text needs truncating";
StringBuilder builder = new StringBuilder(text)
{
    Length = 10
    builder[9] = '\u2026';
    Console.OutputEncoding = Encoding.UTF8;
    Console.WriteLine(builder);
}
```

Sets the Length property to truncate the builder

Modifies the final character to “...”

Makes sure the console will support Unicode

Prints out the builder content

Given how little the initializer is giving you (a single property), I'd at least consider setting the length in a separate statement instead. C# 6 allows you to perform all the initialization you need in a single expression, because you can use the indexer within the object initializer. The following listing demonstrates this in a slightly contrived way.

Listing 10.5 Using an indexer in a `StringBuilder` object initializer

```
string text = "This text needs truncating";
StringBuilder builder = new StringBuilder(text)
{
    Length = 10,
    [9] = '\u2026'
    Console.OutputEncoding = Encoding.UTF8;
    Console.WriteLine(builder);
}
```

Sets the Length property to truncate the builder

Modifies the final character to “...”

Makes sure the console will support Unicode

Prints out the builder content

I deliberately chose to use `StringBuilder` here not because it's the most obvious type containing an indexer but to make it clear that this is an *object* initializer rather than a *collection* initializer.

You might have expected me to use a `Dictionary<, >` of some kind instead, but there's a hidden danger here. If your code is correct, it'll work as you'd expect, but I

recommend sticking to using a collection initializer in most cases. To see why, let's look at an example initializing two dictionaries: one using indexers in an object initializer and one using a collection initializer.

Listing 10.6 Two ways of initializing a dictionary

```
var collectionInitializer = new Dictionary<string, int>    ← Regular collection
{
    { "A", 20 },
    { "B", 30 },
    { "B", 40 }
};

var objectInitializer = new Dictionary<string, int>      ← Object initializer with
{
    ["A"] = 20,
    ["B"] = 30,
    ["B"] = 40
};
```

Superficially, these might look equivalent. When you have no duplicate keys, they're equivalent, and I even prefer the appearance of the object initializer. But the dictionary indexer setter overwrites any existing entry with the same key, whereas the `Add` method throws an exception if the key already exists.

Listing 10.6 deliberately includes the "B" key twice. This is an easy mistake to make, usually as the result of copying and pasting a line and then forgetting to modify the key part. The error won't be caught at compile time in either case, but at least with the collection initializer, it doesn't do the wrong thing silently. If you have any unit tests that execute this piece of code—even if they don't explicitly check the contents of the dictionary—you're likely to find the bug quickly.

Roslyn to the rescue?

Being able to spot this bug at compile time would be better, of course. It should be possible to write an analyzer to spot this problem for both collection and object initializers. For object initializers using an indexer, it's hard to imagine many cases where you'd legitimately want to specify the same constant indexer key multiple times, so popping up a warning seems entirely reasonable.

I don't know of any such analyzer yet, but I hope it'll exist at some point. With that danger cleared, there'd be no reason not to use indexers with dictionaries.

So when should you use an indexer in an object initializer rather than a collection initializer? You should do so in a few reasonably obvious cases, such as the following:

- If you can't use a collection initializer because the type doesn't implement `IEnumerable` or doesn't have suitable `Add` methods. (You can potentially introduce your own `Add` methods as extension methods, however, as you'll see

in the next section.) For example, `ConcurrentDictionary<, >` doesn't have `Add` methods but does have an indexer. It has `TryAdd` and `AddOrUpdate` methods, but those aren't used by the collection initializer. You don't need to worry about concurrent updates to the dictionary while you're in an object initializer, because only the initializing thread has any knowledge of the new dictionary.

- If the indexer and the `Add` method would handle duplicate keys in the same way. Just because dictionaries follow the "throw on add, overwrite in the indexer" pattern doesn't mean that all types do.
- If you're genuinely trying to replace elements rather than adding them. For example, you might be creating one dictionary based on another and then replacing the value corresponding to a particular key.

Less clear-cut cases exist as well in which you need to balance readability against the possibility of the kind of error described previously. Listing 10.7 shows an example of a schemaless entity type with two regular properties, but that otherwise allows arbitrary key/value pairs for its data. You'll then look at the options for how you might initialize an instance.

Listing 10.7 A schemaless entity type with key properties

```
public sealed class SchemalessEntity
    : IEnumerable<KeyValuePair<string, object>>
{
    private readonly IDictionary<string, object> properties =
        new Dictionary<string, object>();

    public string Key { get; set; }
    public string ParentKey { get; set; }

    public object this[string propertyKey]
    {
        get { return properties[propertyKey]; }
        set { properties[propertyKey] = value; }
    }

    public void Add(string propertyKey, object value)
    {
        properties.Add(propertyKey, value);
    }

    public IEnumerator<KeyValuePair<string, object>> GetEnumerator() =>
        properties.GetEnumerator();

    IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
}
```

Let's consider two ways of initializing an entity for which you want to specify a parent key, the new entity's key, and two properties (a name and location, just as simple strings). You can use a collection initializer but then set the other properties afterward

or do the whole thing with an object initializer but risk typos in the keys. The following listing demonstrates both options.

Listing 10.8 Two ways of initializing a SchemalessEntity

```

SchemalessEntity parent = new SchemalessEntity { Key = "parent-key" };
SchemalessEntity child1 = new SchemalessEntity
{
    { "name", "Jon Skeet" },
    { "location", "Reading, UK" }
};
child1.Key = "child-key";
child1.ParentKey = parent.Key;           | Specifies key
                                         | properties separately

SchemalessEntity child2 = new SchemalessEntity
{
    Key = "child-key",                  | Specifies key properties
    ParentKey = parent.Key,            | in an object initializer
    ["name"] = "Jon Skeet",
    ["location"] = "Reading, UK"       | Specifies data properties
                                         | using indexers
};

```

Which of these approaches is better? The second looks a lot cleaner to me. I'd typically extract the name and location keys into string constants anyway, at which point the risk of accidentally using duplicate keys is at least reduced.

If you're in control of a type like this, you can add extra members to allow you to use a collection initializer. You could add a `Properties` property that either exposes the dictionary directly or exposes a view over it. At that point, you could use a collection initializer to initialize `Properties` within an object initializer that also sets `Key` and `ParentKey`. Alternatively, you could provide a constructor that accepts the key and parent key, at which point you can make an explicit constructor call with those values and then specify the name and location properties with a collection initializer.

This may feel like a huge amount of detail for a choice between using indexers in an object initializer or using a collection initializer as in previous versions. The point is that the choice is yours to make: no book will be able to give you simple rules to follow that give you a best answer in every case. Be aware of the pros and cons, and apply your own judgment.

10.2.2 Using extension methods in collection initializers

A second change in C# 6 related to object and collection initializers concerns which methods are available in collection initializers. As a reminder, two conditions must be met in order to use a collection initializer with a type:

- The type must implement `IEnumerable`. I've found this to be an annoying restriction; sometimes I implement `IEnumerable` solely so I can use the type in collection initializers. But it is what it is. This restriction hasn't changed in C# 6.
- There must be a suitable `Add` method for every element in the collection initializer. Any elements that aren't in curly braces are assumed to correspond to

single-argument calls to Add methods. When multiple arguments are required, they must be in curly braces.

Occasionally, this can be a little restrictive. Sometimes you want to easily create a collection in a way that the Add methods supplied by the collection itself don't support. The preceding conditions are still true in C# 6, but the definition of "suitable" in the second condition now includes extension methods. In some ways, this has simplified the transformation. Here's a declaration using a collection initializer:

```
List<string> strings = new List<string>
{
    10,
    "hello",
    { 20, 3 }
};
```

That declaration is essentially equivalent to this:

```
List<string> strings = new List<string>();
strings.Add(10);
strings.Add("hello");
strings.Add(20, 3);
```

The normal overload resolution is applied to work out what each of those method calls means. If that fails, the collection initializer won't compile. With just the regular List<T>, the preceding code won't compile, but if you add a single extension method it will:

```
public static class StringListExtensions
{
    public static void Add(
        this List<string> list, int value, int count = 1)
    {
        list.AddRange(Enumerable.Repeat(value.ToString(), count));
    }
}
```

With this in place, the first and last calls to Add in our earlier code end up calling the extension method. The list ends up with five elements ("10", "hello", "20", "20", "20"), because the last Add call adds three elements. This is an unusual extension method, but it helps demonstrate three points:

- Extension methods can be used in collection initializers, which is the whole point of this section of the book.
- This isn't a generic extension method; it works for only List<string>. This is a kind of specialization that couldn't be performed in List<T> itself. (Generic extension methods are fine, too, so long as the type arguments can be inferred.)
- Optional parameters can be used in the extension methods; our first call to Add will effectively be compiled to Add(10, 1) because of the default value of the second parameter.

Now that you know what you can do, let's take a closer look at where it makes sense to use this feature.

CREATING OTHER GENERAL-PURPOSE ADD SIGNATURES

One technique I've found useful in my work with Protocol Buffers is to create Add methods accepting collections. This process is like using AddRange but it can be used in collection initializers. This is particularly useful within object initializers in which the property you're initializing is read-only but you want to add the results of a LINQ query.

For example, consider a Person class with a read-only Contacts property that you want to populate with all the contacts from another list who live in Reading. In Protocol Buffers, the Contacts property would be of type RepeatedField<Person>, and RepeatedField<T> has the appropriate Add method, allowing you to use a collection initializer:

```
Person jon = new Person
{
    Name = "Jon",
    Contacts = { allContacts.Where(c => c.Town == "Reading") }
};
```

It can take a little getting used to, but then it's extremely useful and certainly beats having to call `jon.Contacts.AddRange(...)` separately. But what if you weren't using Protocol Buffers, and Contacts was exposed only as `List<Person>` instead? With C# 6, that's not a problem: you can create an extension method for `List<T>` that adds an overload of Add accepting an `IEnumerable<T>` and calling `AddRange` with it, as shown in the following listing.

Listing 10.9 Exposing explicit interface implementations via extension methods

```
static class ListExtensions
{
    public static void Add<T>(this List<T> list, IEnumerable<T> collection)
    {
        list.AddRange(collection);
    }
}
```

With that extension method in place, the earlier code works fine even with `List<T>`. If you wanted to be broader still, you could write an extension method targeting `IList<T>` instead, although if you went down that route, you'd need to write the loop within the method body because `IList<T>` doesn't have an `AddRange` method.

CREATING SPECIALIZED ADD SIGNATURES

Suppose you have a Person class, as shown earlier, with a Name property, and within one area of code you do a lot of work with `Dictionary<string, Person>` objects, always indexing the Person objects by name. Adding entries to the dictionary with a simple call to `dictionary.Add(person)` can be convenient, but `Dictionary`

`<string, Person>` doesn't, as a type, know that you're indexing by name. What are your choices?

You could create a class derived from `Dictionary<string, Person>` and add an `Add(Person)` method to it. That doesn't appeal to me, because you're not specializing the behavior of the dictionary in any meaningful way; you're just making it more convenient to use.

You could create a more general class implementing `IDictionary< TKey, TValue >` that accepts a delegate explaining the mapping from `TValue` to `TKey` and implement that via composition. That could be useful but may be overkill for this one task. Finally, you could create an extension method for this one specific case, as shown in the following listing.

Listing 10.10 Adding a type-argument-specific Add method for dictionaries

```
static class PersonDictionaryExtensions
{
    public static void Add(
        this Dictionary<string, Person> dictionary, Person person)
    {
        dictionary.Add(person.Name, person);
    }
}
```

That already would've been a good option before C# 6, but the combination of using the `using static` feature to limit the way extension methods are imported along with the use of extension methods in collection initializers makes it more compelling. You can then initialize a dictionary without any repetition of the name:

```
var dictionary = new Dictionary<string, Person>
{
    { new Person { Name = "Jon" } },
    { new Person { Name = "Holly" } }
};
```

An important point here is how you've specialized the API for one particular combination of type arguments to `Dictionary<, >` but without changing the type of object you're creating. No other code needs to be aware of the specialization here, because it's only superficial; it exists only for our convenience rather than being part of an object's inherent behavior.

NOTE This approach has downsides as well, one of which is that nothing prevents an entry from being added by using something other than a person's name. As ever, I encourage you to think through the pros and cons for yourself; don't blindly trust my advice or anyone else's.

REEXPOSING EXISTING METHODS “HIDDEN” BY EXPLICIT INTERFACE IMPLEMENTATION

In section 10.2.1, I used `ConcurrentDictionary<, >` as an example of where you might want to use an indexer instead of a collection initializer. Without any extra

help, you can't use a collection initializer because no `Add` method is exposed. But `ConcurrentDictionary<, >` does have an `Add` method; it's just that it uses explicit interface implementation to implement `IDictionary<, >.Add`. Usually, if you want to access a member that uses explicit interface implementation, you have to cast to the interface—but you can't do that in a collection initializer. Instead, you can expose an extension method, as shown in the following listing.

Listing 10.11 Exposing explicit interface implementations via extension methods

```
public static class DictionaryExtensions
{
    public static void Add<TKey, TValue>(
        this IDictionary<TKey, TValue> dictionary,
        TKey key, TValue value)
    {
        dictionary.Add(key, value);
    }
}
```

At first glance, this looks completely pointless. It's an extension method to call a method with exactly the same signature. But this effectively works around explicit interface implementation, making the `Add` method always available, including in collection initializers. You can now use a collection initializer for `ConcurrentDictionary<, >`:

```
var dictionary = new ConcurrentDictionary<string, int>
{
    { "x", 10 },
    { "y", 20 }
};
```

This should be used cautiously, of course. When a method is obscured by explicit interface implementation, that's often meant to discourage you from calling it without a certain amount of care. This is where the ability to selectively import extension methods with `using static` is useful: you could have a namespace of static classes with extension methods that are meant to be used only selectively and import just the relevant class in each case. Unfortunately, it still exposes the `Add` method to the rest of the code in the same class, but again you need to weigh whether that's worse than the alternatives.

The extension method in listing 10.11 is broad, extending all dictionaries. You could decide to target only `ConcurrentDictionary<, >` instead to avoid inadvertently using an explicitly implemented `Add` method from another dictionary type.

10.2.3 Test code vs. production code

You've probably noticed a lot of caveats in this section. Few clear-cut cases exist that enable you to “definitely use it here” with respect to these features. But most of the downsides I've noted are in terms of areas where the feature is convenient in one piece of code but you don't want it infecting other places.

My experience is that object and collection initializers are usually used in two places:

- Static initializers for collections that'll never be modified after type initialization
- Test code

The concerns around exposure and correctness still apply for static initializers but much less so for test code. If you decide that in your test assemblies it's handy to have `Add` extension methods to make collection initializers simpler, that's fine. It won't impact your production code at all. Likewise, if you use indexers in your collection initializers for tests and accidentally set the same key twice, chances are high that your tests will fail. Again, the downside is minimized.

This isn't a distinction that affects only this pair of features. Test code should still be of high quality, but how you measure that quality and the impact of making any particular trade-off is different for test code as compared to production code, particularly for public APIs.

The addition of extension methods as part of LINQ encouraged a more fluent approach to composing multiple operations. Instead of using multiple statements, in many cases it's now idiomatic to chain multiple method calls together in a single statement. That's what LINQ queries end up doing all the time, but it became a more idiomatic pattern with APIs such as LINQ to XML. This can lead to the same problem we've had for a long time when chaining property accesses together: everything breaks as soon as you encounter a null value. C# 6 allows you terminate one of these chains safely at that point instead of the code blowing up with an exception.

10.3 **The null conditional operator**

I'm not going to go into the merits of nullity, but it's something we often have to live with, along with complex object models with properties several levels deep. The C# language team has been thinking for a long time about making nullity easier to work with. Some of that work is still in progress, but C# 6 has taken one step along the way. Again, it can make your code much shorter and simpler, expressing how you want to handle nullity without having to repeat expressions everywhere.

10.3.1 **Simple and safe property dereferencing**

As a working example, let's suppose you have a `Customer` type with a `Profile` property that has a `DefaultShippingAddress` property, which has a `Town` property. Now let's suppose you want to find all customers within a collection whose default shipping address has `Reading` as a town name. Without worrying about nullity, you could use this:

```
var readingCustomers = allCustomers
    .Where(c => c.Profile.DefaultShippingAddress.Town == "Reading");
```

That works fine if you know that every customer has a profile, every profile has a default shipping address, and every address has a town. But what if any of those is null? You'll end up with a `NullReferenceException` when you probably just want

to exclude that customer from the results. Previously, you'd have to rewrite this as something horrible, checking each property for nullity one at a time by using the short-circuiting `&&` operator:

```
var readingCustomers = allCustomers
    .Where(c => c.Profile != null &&
              c.Profile.DefaultShippingAddress != null &&
              c.Profile.DefaultShippingAddress.Town == "Reading");
```

Yeesh. So much repetition. It gets even worse if you need to make a method call at the end rather than using `==` (which already handles null correctly, at least for references; see section 10.3.3 for possible surprises). So how does C# 6 improve this? It introduces the *null conditional* `?.` operator, which is a short-circuiting operator that stops if the expression evaluates to null. A null-safe version of the query is as follows:

```
var readingCustomers = allCustomers
    .Where(c => c.Profile?.DefaultShippingAddress?.Town == "Reading");
```

This is exactly the same as our first version but with two uses of the null-conditional operator. If either `c.Profile` or `c.Profile.DefaultShippingAddress` is null, the whole expression on the left side of `==` evaluates to null. You may be asking yourself why you have only two uses, when four things are potentially null:

- `c`
- `c.Profile`
- `c.Profile.DefaultShippingAddress`
- `c.Profile.DefaultShippingAddress.Town`

I've assumed that all the elements of `allCustomers` are non-null references. If you needed to handle the possibility of null elements there, you could use `c?.Profile` at the start instead. That covers the first bullet; the `==` operator already handles null operands, so you don't need to worry about the last bullet.

10.3.2 The null conditional operator in more detail

This brief example shows only properties, but the null conditional operator can also be used to access methods, fields, and indexers. The basic rule is that when a null conditional operator is encountered, the compiler injects a nullity check on the value to the left of the `?`. If the value is null, evaluation stops and the result of the overall expression is null. Otherwise, evaluation continues with the property, method, field, or index access to the right of the `?` without reevaluating the first part of the expression. If the type of the overall expression would be a non-nullable value type without the null conditional operator, it becomes the nullable equivalent if a null conditional operator is involved anywhere in the sequence.

The overall expression here—the part where evaluation stops abruptly if a null value is encountered—is basically the sequence of property, field, indexer, and method access involved. Other operators, such as comparisons, break the sequence

because of precedence rules. To demonstrate this, let's have a closer look at the condition for the `Where` method in section 10.3.1. Our lambda expression was as follows:

```
c => c.Profile?.DefaultShippingAddress?.Town == "Reading"
```

The compiler treats this roughly as if you'd written this:

```
string result;
var tmp1 = c.Profile;
if (tmp1 == null)
{
    result = null;
}
else
{
    var tmp2 = tmp1.DefaultShippingAddress;
    if (tmp2 == null)
    {
        result = null;
    }
    else
    {
        result = tmp2.Town;
    }
}
return result == "Reading";
```

Notice how each property access (which I've highlighted in bold) occurs only once. In our pre-C# 6 version checking for null, you'd potentially evaluate `c.Profile` three times and `c.Profile.DefaultShippingAddress` twice. If those evaluations depended on data being mutated by other threads, you could be in trouble: you could pass the first two nullity tests and still fail with a `NullReferenceException`. The C# code is safer and more efficient because you're evaluating everything only once.

10.3.3 Handling Boolean comparisons

Currently, you're still performing the comparison at the end with the `==` operator; that isn't short-circuited away if anything is null. Suppose you want to use the `Equals` method instead and write this:

```
c => c.Profile?.DefaultShippingAddress?.Town?.Equals("Reading")
```

Unfortunately, this doesn't compile. You've added a third null conditional operator, so you don't call `Equals` if you have a shipping address with a `Town` property of null. But now the overall result is `Nullable<bool>` instead of `bool`, which means our lambda expression isn't suitable for the `Where` method yet.

This is a pretty common occurrence with the null conditional operator. Anytime you use the null conditional operator in any kind of condition, you need to consider three possibilities:

- Every part of the expression is evaluated, and the result is true.
- Every part of the expression is evaluated, and the result is false.
- The expression short-circuited because of a null value, and the result is null.

Usually, you want to collapse those three possibilities down to two by making the third option map to a true or false result. There are two common ways of doing this: comparing against a `bool` constant or using the null coalescing `??` operator.

Language design choices for nullable Boolean comparisons

The behavior of `bool?` in comparisons with non-nullable values caused concern for the language designers in the C# 2 time frame. The fact that `x == true` and `x != false` are both valid but with different meanings if `x` is a `bool?` variable can be pretty surprising. (If `x` is null, `x == true` evaluates to `false`, and `x != false` evaluates to `true`.)

Was it the right design choice? Maybe. Often all the choices available are unpleasant in one respect or other. It won't change now, though, so it's best to be aware of it and write code as clearly as possible for readers who may be less aware.

To simplify our example, let's suppose you already have a variable called `name` containing the relevant string value, but it can be null. You want to write an `if` statement and execute the body of the statement if the town is X based on the `Equals` method. This is the simplest way of demonstrating a condition: in real life, you could be conditionally accessing a Boolean property, for example. Table 10.1 shows the options you can use depending on whether you also want to enter the body of the statement if `name` is null.

Table 10.1 Options for performing Boolean comparisons using the null conditional operator

You don't want to enter the body if <code>name</code> is null	You do want to enter the body if <code>name</code> is null
<pre>if (name?.Equals("X") ?? false) if (name?.Equals("X") == true)</pre>	<pre>if (name?.Equals("X") ?? true) if (name?.Equals("X") != false)</pre>

I prefer the null coalescing operator approach; I read it as “try to perform the comparison, but default to the value after the `??` if you have to stop early.” After you understand that the type of the expression (`name?.Equals("X")` in this case) is `Nullable<bool>`, nothing else is new here. It just so happens that you're much more likely to come up against this case than you were before the null conditional operator became available.

10.3.4 Indexers and the null conditional operator

As I mentioned earlier, the null conditional operator works for indexers as well as for fields, properties, and methods. The syntax is again just adding a question mark, but this time before the opening square bracket. This works for array access as well as

user-defined indexers, and again the result type becomes nullable if it would otherwise be a non-nullable value type. Here's a simple example:

```
int[] array = null;
int? firstElement = array?[0];
```

There's not a lot more to say about how the null-conditional operator works with indexers; it's as simple as that. I haven't found this to be nearly as useful as working with properties and methods, but it's still good to know that it's there, as much for consistency as anything else.

10.3.5 Working effectively with the null conditional operator

You've already seen that the null conditional operator is useful when working with object models with properties that may or may not be null, but other compelling use cases exist. We'll look at two of them here, but this isn't an exhaustive list, and you may come up with additional novel uses yourself.

SAFE AND CONVENIENT EVENT RAISING

The pattern for raising an event safely even in the face of multiple threads has been well known for many years. For example, to raise a field-like `Click` event of type `EventHandler`, you'd write code like this:

```
EventHandler handler = Click;
if (handler != null)
{
    handler(this, EventArgs.Empty);
}
```

Two aspects are important here:

- You're not just calling `Click(this, EventArgs.Empty)`, because `Click` might be null. (That would be the case if no handler was subscribed to the event.)
- You're copying the value of the `Click` field to a local variable first so that even if it changes in another thread after you've checked for nullity, you still have a non-null reference. You may invoke a "slightly old" (just unsubscribed) event handler, but that's a reasonable race condition.

So far, so good—but so long-winded. The null conditional operator comes to the rescue, however. It can't be used for the shorthand style of delegate invocation of `handler(...)`, but you can use it to conditionally call the `Invoke` method and all in a single line:

```
Click?.Invoke(this, EventArgs.Empty);
```

If this is the only line in your method (`OnClick` or similar), this has the compound benefit that now it's a single-expression body, so it can be written as an expression-bodied method. It's just as safe as the earlier pattern but a good deal more concise.

MAKING THE MOST OF NULL-RETURNING APIs

In chapter 9, I talked about logging and how interpolated string literals don't help in terms of performance. But they can be cleanly combined with the null conditional operator if you have a logging API designed with that pattern in mind. For example, suppose you have a logger API along the lines of that shown in the next listing.

Listing 10.12 Sketch of a null-conditional-friendly logging API

```
public interface ILogger
{
    IAActiveLogger Debug { get; }
    IAActiveLogger Info { get; }
    IAActiveLogger Warning { get; }
    IAActiveLogger Error { get; }
}

public interface IAActiveLogger
{
    void Log(string message);
}
```

Annotations:

- ILogger: Interface returned by GetLog methods and so on
- IAActiveLogger: Properties returning null when the log is disabled
- IAActiveLogger: Interface representing an enabled log sink

This is only a sketch; a full logging API would have much more to it. But by separating the step of getting an active logger at a particular log level from the step of performing the logging, you can write efficient and informative logging:

```
logger.Debug?.Log($"Received request for URL {request.Url}");
```

If debug logging is disabled, you never get as far as formatting the interpolated string literal, and you can determine that without creating a single object. If debug logging is enabled, the interpolated string literal will be evaluated and passed on to the `Log` method as usual. Without getting too misty eyed, this is the sort of thing that makes me love the way C# has evolved.

Of course, you need the logging API to handle this in an appropriate way first. If whichever logging API you're using doesn't have anything like this, extension methods might help you out.

A lot of the reflection APIs return null at appropriate times, and LINQ's `FirstOrDefault` (and similar) methods can work well with the null-conditional operator. Likewise, LINQ to XML has many methods that return null if they can't find what you're asking for. For example, suppose you have an XML element with an optional `<author>` element that may or may not have a `name` attribute. You can easily retrieve the author name with either of these two statements:

```
string authorName = book.Element("author")?.Attribute("name")?.Value;
string authorName = (string) book.Element("author")?.Attribute("name");
```

The first of these uses the null conditional operator twice: once to access the attribute of the element and once to access the value of the attribute. The second approach uses the way that LINQ to XML already embraces nullity in its explicit conversion operators.

10.3.6 Limitations of the null conditional operator

Beyond occasionally having to deal with nullable value types when previously you were using only non-nullable values, there are few unpleasant surprises with the null conditional operator. The only thing that might surprise you is that the result of the expression is always classified as a value rather than a variable. The upshot of this is that you can't use the null-conditional operator as the left side of an assignment. For example, the following are all invalid:

```
person?.Name = "";
stats?.RequestCount++;
array?[index] = 10;
```

In those cases, you need to use the old-fashioned `if` statement. My experience is that this limitation is rarely an issue.

The null conditional operator is great for avoiding `NullReferenceException`, but sometimes exceptions happen for more reasonable causes, and you need to be able to handle them. Exception filters represent the first change to the structure of a catch block since C# was first introduced.

10.4 Exception filters

Our final feature in this chapter is a little embarrassing: it's C# playing catch-up with VB. Yes, VB has had exception filters forever, but they were introduced only in C# 6. This is another feature you may rarely use, but it's an interesting peek into the guts of the CLR. The basic premise is that you can now write catch blocks that only *sometimes* catch an exception based whether a filter expression returns `true` or `false`. If it returns `true`, the exception is caught. If it returns `false`, the catch block is ignored.

As an example, imagine you're performing a web operation and know that the server you're connecting to is sometimes offline. If you fail to connect to it, you have another option, but any other kind of failure should result in an exception bubbling up in the normal way. Prior to C# 6, you'd have to catch the exception and rethrow it if didn't have the right status:

```
try
{
    ...
}
catch (WebException e)
{
    if (e.Status != WebExceptionStatus.ConnectFailure)
    {
        throw;
    }
    ...
}
```

The diagram shows the flow of control. It starts with a `try` block containing some placeholder code (`...`). An arrow points from this block to a bracket labeled "Attempts the web operation", which covers the entire `try` block. Below it is a `catch` block for `WebException`. Inside the `catch` block, there is an `if` statement that checks the `e.Status` against `WebExceptionStatus.ConnectFailure`. If the condition is false, the `throw` statement is executed, which is then connected by a vertical line to the "Handles the connection failure" bracket. If the condition is true, the code continues after the `if` block, which is also covered by the "Handles the connection failure" bracket.

With an exception filter, if you don't want to handle an exception, you don't catch it; you filter it away from your `catch` block to start with:

```
try
{
    ...
} Attempts the web operation
catch (WebException e)
    when (e.Status == WebExceptionStatus.ConnectFailure) Catches only connection failures
{
    ...
} Handles the connection failure
```

Beyond specific cases like this, I can see exception filters being useful in two generic use cases: retry and logging. In a retry loop, you typically want to catch the exception only if you're going to retry the operation (if it meets certain criteria and you haven't run out of attempts); in a logging scenario, you may never want to catch the exception but log it while it's in-flight, so to speak. Before going into more details of the concrete use cases, let's see what the feature looks like in code and how it behaves.

10.4.1 Syntax and semantics of exception filters

Our first full example, shown in the following listing, is simple: it loops over a set of messages and throws an exception for each of them. You have an exception filter that will catch exceptions only when the message contains the word `catch`. The exception filter is highlighted in bold.

Listing 10.13 Throwing three exceptions and catching two of them

```
string[] messages =
{
    "You can catch this",
    "You can catch this too",
    "This won't be caught"
};
foreach (string message in messages) Loops outside the try/catch statement once per message
{
    try
    {
        throw new Exception(message);
    } Throws an exception with a different message each time
    catch (Exception e)
        when (e.Message.Contains("catch")) Catches the exception only if it contains "catch"
    {
        Console.WriteLine($"Caught '{e.Message}'");
    } Writes out the message of the caught exception
}
```

The output is two lines for the caught exceptions:

```
Caught 'You can catch this'
Caught 'You can catch this too'
```

Output for the uncaught exception is a message of This won't be caught. (Exactly what that looks like depends on how you run the code, but it's a normal unhandled exception.)

Syntactically, that's all there is to exception filters: the contextual keyword `when` followed by an expression in parentheses that can use the exception variable declared in the `catch` clause and must evaluate to a Boolean value. The semantics may not be quite what you expect, though.

THE TWO-PASS EXCEPTION MODEL

You're probably used to the idea of the CLR unwinding the stack as an exception "bubbles up" until it's caught. What's more surprising is exactly how this happens. The process is more complicated than you may expect using a *two-pass model*.¹ This model uses the following steps:

- The exception is thrown, and the *first pass* starts.
- The CLR walks down the stack, trying to find which `catch` block will handle the exception. (We'll call this the *handling catch block* as shorthand, but that's not official terminology.)
- Only `catch` blocks with compatible exception types are considered.
- If a `catch` block has an exception filter, the filter is executed; if the filter returns `false`, this `catch` block *won't* handle the exception.
- A `catch` block without an exception filter is equivalent to one with an exception filter that returns `true`.
- Now that the handling `catch` block has been determined, the *second pass* starts:
- The CLR unwinds the stack from the point at which the exception was thrown as far as the `catch` block that has been determined.
- Any `finally` blocks encountered while unwinding the stack are executed. (This doesn't include any `finally` block associated with the handling `catch` block.)
- The handling `catch` block is executed.
- The `finally` statement associated with the handling `catch` block is executed, if there is one.

Listing 10.14 shows a concrete example of all of this with three important methods: `Bottom`, `Middle`, and `Top`. `Bottom` calls `Middle` and `Middle` calls `Top`, so the stack ends up being self-describing. The `Main` method calls `Bottom` to start the ball rolling. Please don't be daunted by the length of this code; it's not doing anything massively complicated. Again, the exception filters are highlighted in bold. The `LogAndReturn` method is just a convenient way to trace the execution. It's used by exception filters to log a particular method and then return the specified value to say whether the exception should be caught.

¹ I don't know the origins of this model for exception processing. I suspect it maps onto the Windows Structured Exception Handling (often abbreviated to SEH) mechanism in a straightforward way, but this is deeper into the CLR than I like to venture.

Listing 10.14 A three-level demonstration of exception filtering

```

static bool LogAndReturn(string message, bool result)
{
    Console.WriteLine(message);
    return result;
}

static void Top()
{
    try
    {
        throw new Exception();
    }
    finally
    {
        Console.WriteLine("Top finally");
    }
}

static void Middle()
{
    try
    {
        Top();
    }
    catch (Exception e)
        when (LogAndReturn("Middle filter", false))
    {
        Console.WriteLine("Caught in middle");
    }
    finally
    {
        Console.WriteLine("Middle finally");
    }
}

static void Bottom()
{
    try
    {
        Middle();
    }
    catch (IOException e)
        when (LogAndReturn("Never called", true))
    {
    }
    catch (Exception e)
        when (LogAndReturn("Bottom filter", true))
    {
        Console.WriteLine("Caught in Bottom");
    }
}

```

Convenience method called by exception filters

Finally block (no catch) executed on the second pass

Exception filter that never catches
This never prints, because the filter returns false.

Finally block executed on the second pass

Exception filter that's never called—wrong exception type

Exception filter that always catches
This is printed, because you catch the exception here.

```
static void Main()
{
    Bottom();
}
```

Phew! With the description earlier and the annotations in the listing, you have enough information to work out what the output will be. We'll walk through it to make sure it's really clear. First, let's look at what's printed:

```
Middle filter
Bottom filter
Top finally
Middle finally
Caught in Bottom
```

Figure 10.2 shows this process. In each step, the left side shows the stack (ignoring Main), the middle part describes what's happening, and the right side shows any output from that step.

Stack	Explanation of progress	Output
Top Middle Bottom	Bang! Top throws exception. First pass starts.	
Top Middle Bottom	Walking down the stack: exception filter in middle evaluated. It returns false, so keep going.	Middle filter
Top Middle Bottom	Walking down the stack: exception filter in bottom evaluated. It returns true, so first pass is complete! Second pass starts.	Bottom filter
Top Middle Bottom	Finally block in top executes. Stack unwinds.	Top finally
Middle Bottom	Finally block in middle executes. Stack unwinds.	Middle finally
Bottom	Catch block in bottom executes. Finished!	Caught in bottom

Figure 10.2 Execution flow of listing 10.14

Security impact of the two-pass model

The execution timing of `finally` blocks affects `using` and `lock` statements, too. This has an important implication of what you can use `try/finally` or `using` for if you're writing code that may be executed in an environment that may contain hostile code. If your method may be called by code you don't trust, and you allow exceptions to escape from that method, then the caller can use an exception filter to execute code before your `finally` block executes.

All of this means you shouldn't use `finally` for anything security sensitive. For example, if your `try` block enters a more privileged state and you're relying on a `finally` block to return to a less privileged state, other code could execute while you're still in that privileged state. A lot of code doesn't need to worry about this sort of thing—it's always running under friendly conditions—but you should definitely be aware of it. If you're concerned, you could use an empty `catch` block with a filter that removes the privilege and returns `false` (so the exception isn't caught), but that's not something I'd want to do regularly.

CATCHING THE SAME EXCEPTION TYPE MULTIPLE TIMES

In the past, it was always an error to specify the same exception type in multiple `catch` blocks for the same `try` block. It didn't make any sense, because the second block would never be reached. With exception filters, it makes a lot more sense.

To demonstrate this, let's expand our initial `WebException` example. Suppose you're fetching web content based on a URL provided by a user. You might want to handle a connection failure in one way, a name resolution failure in a different way, and let any other kind of exception bubble up to a higher-level `catch` block. With exception filters, you can do that simply:

```
try
{
    ...
}
catch (WebException e)
    when (e.Status == WebExceptionStatus.ConnectFailure)
{
    ...
}
catch (WebException e)
    when (e.Status == WebExceptionStatus.NameResolutionFailure)
{
    ...
}
```

If you wanted to handle all other `WebExceptions` at the same level, it'd be valid to have a general `catch (WebException e) { ... }` block with no exception filter after the two status-specific ones.

Now that you know how exception filters work, let's return to the two generic examples I gave earlier. These aren't the only uses, but they should help you recognize other, similar situations. Let's start with retrying.

10.4.2 Retrying operations

As cloud computing becomes more prevalent, we're generally becoming more aware of operations that can fail and the need to think about what effect we want that failure to have on our code. For remote operations—web service calls and database operations, for example—there are sometimes transient failures that are perfectly safe to retry.

Keep track of your retry policies

Although being able to retry like this is useful, it's worth being aware of every layer of your code that might be attempting to retry a failed operation. If you have multiple layers of abstraction each trying to be nice and transparently retrying a failure that might be transient, you can end up delaying logging a real failure for a long time. In short, it's a pattern that doesn't compose well with itself.

If you control the whole stack for an application, you should think about where you want the retry to occur. If you're responsible for only one aspect of it, you should consider making the retry configurable so that a developer who does control the whole stack can determine whether your layer is where they want retries to occur.

Production retry handling is somewhat complicated. You may need complicated heuristics to determine when and how long to retry for and an element of randomness on the delays between attempts to avoid retrying clients getting in sync with each other. Listing 10.15 provides a hugely simplified version² to avoid distracting you from the exception filter aspects.

All your code needs to know is the following:

- What operation you're trying to execute
- How many times you're willing to try it

At that point, using an exception filter to catch exceptions only when you're going to retry the operation, the code is straightforward.

Listing 10.15 A simple retry loop

```
static T Retry<T>(Func<T> operation, int attempts)
{
    while (true)
    {
        try
        {
            attempts--;
            return operation();
        }
        catch (Exception e) when (attempts > 0)
```

² At a bare minimum, I'd expect any real-world retry mechanism to accept a filter to check which failures are retriable and a delay between calls.

```
        {
            Console.WriteLine($"Failed: {e}");
            Console.WriteLine($"Attempts left: {attempts}");
            Thread.Sleep(5000);
        }
    }
}
```

Although `while(true)` loops are rarely a good idea, this one makes sense. You could write a loop with a condition based on `retryCount`, but the exception filter effectively already provides that, so it'd be misleading. Also, the end of the loop would then be reachable from the compiler's standpoint, so it wouldn't compile without a `return` or `throw` statement at the end of the method.

When this is in place, calling it to achieve a retry is simple:

```
Func<DateTime> temporamentalCall = () =>
{
    DateTime utcNow = DateTime.UtcNow;
    if (utcNow.Second < 20)
    {
        throw new Exception("I don't like the start of a minute");
    }
    return utcNow;
};

var result = Retry(temporamentalCall, 3);
Console.WriteLine(result);
```

Usually, this will return a result immediately. Sometimes, if you execute it at about 10 seconds into a minute, it'll fail a couple of times and then succeed. Sometimes, if you execute it right at the start of a minute, it'll fail a couple of times, catching the exception and logging it, and then fail a third time, at which point the exception won't be caught.

10.4.3 Logging as a side effect

Our second example is a way of logging exceptions in-flight. I realize I've used logging to demonstrate many of the C# 6 features, but this is a coincidence. I don't believe the C# team decided that they'd target logging specifically for this release; it just works well as a familiar scenario.

The subject of exactly how and where it makes sense to log exceptions is a matter of much debate, and I don't intend to enter that debate here. Instead, I'll assert that at least sometimes, it's useful to log an exception within one method call even if it's going to be caught (and possibly logged a second time) somewhere further down the stack.

You can use exception filters to log the exception in a way that doesn't disturb the execution flow in any other way. All you need is an exception filter that calls a method to log the exception and then returns `false` to indicate that you don't really want to catch the exception. The following listing demonstrates this in a `Main` method that

will still lead to the process completing with an error code, but only after it has logged the exception with a timestamp.

Listing 10.16 Logging in a filter

```
static void Main()
{
    try
    {
        UnreliableMethod();
    }
    catch (Exception e) when (Log(e))
    {
    }
}

static void UnreliableMethod()
{
    throw new Exception("Bang!");
}

static bool Log(Exception e)
{
    Console.WriteLine($"{DateTime.UtcNow}: {e.GetType()} {e.Message}");
    return false;
}
```

This listing is in many ways just a variation of listing 10.14, in which we used logging to investigate the semantics of the two-pass exception system. In this case, you're never catching the exception in the filter; the whole `try/catch` and filter exist only for the side effect of logging.

10.4.4 Individual, case-specific exception filters

In addition to those generic examples, specific business logic sometimes requires some exceptions to be caught and others to propagate further. If you doubt that this is ever useful, consider whether you always catch `Exception` or whether you tend to catch specific exception types like `IOException` or `SqlException`. Consider the following block:

```
catch (IOException e)
{
    ...
}
```

You can think of that block as being broadly equivalent to this:

```
catch (Exception tmp) when (tmp is IOException)
{
    IOException e = (IOException) tmp;
    ...
}
```

Exception filters in C# 6 are a generalization of that. Often, the relevant information isn't in the type but is exposed in some other way. Take `SqlException`, for example; it has a `Number` property corresponding to an underlying cause. It'd be far from unreasonable to handle some SQL failures in one way and others in a different way. Getting the underlying HTTP status from a `WebException` is slightly tricky because of the API, but again, you may well want to handle a 404 (Not Found) response differently than a 500 (Internal Error).

One word of caution: I strongly urge you *not* to filter based on the exception message (other than for experimental purposes, as I did in listing 10.13). Exception messages aren't generally seen as having to stay stable between releases, and they may well be localized, depending on the source. Code that behaves differently based on a particular exception message is fragile.

10.4.5 Why not just throw?

You may be wondering what all the fuss is about. We've always been able to rethrow exceptions, after all. Code using an exception filter like this

```
catch (Exception e) when (condition)
{
    ...
}
```

isn't very different from this:

```
catch (Exception e)
{
    if (!condition)
    {
        throw;
    }
    ...
}
```

Does this really meet the high bar for a new language feature? It's arguable.

There *are* differences between the two pieces of code: you've already seen that the timing of when `condition` is evaluated changes relative to any `finally` blocks higher up the call stack. Additionally, although a simple `throw` statement does preserve the original stack trace for the most part, subtle differences can exist, particularly in the stack frame where the exception is caught and rethrown. That could certainly make the difference between diagnosing an error being simple and it being painful.

I doubt that exception filters will massively transform many developers' lives. They're not something I miss when I have to work on a C# 5 codebase, unlike expression-bodied members and interpolated string literals, for example, but they're still nice to have.

Of the features described in this chapter, using `static` and the null conditional operator are certainly the ones I use most. They're applicable in a broad range of cases and can sometimes make the code radically more readable. (In particular, if you

have code that deals with a lot of constants defined elsewhere, using `static` can make all the difference in terms of readability.)

One aspect that's common to the null conditional operator and the object/collection initializer improvements is the ability to express a complex operation in a single expression. This reinforces the benefits that object/collection initializers introduced back in C# 3: it allows expressions to be used for field initialization or method arguments that might otherwise have had to be computed separately and less conveniently.

Summary

- `using static` directives allow your code to refer to static type members (usually constants or methods) without specifying the type name again.
- `using static` also imports all extension methods from the specified type, so you don't need to import all the extension methods from a namespace.
- Changes to extension method importing mean that converting a regular static method into an extension method is no longer a backward-compatible change in all cases.
- Collection initializers can now use `Add` extension methods as well as those defined on the collection type being initialized.
- Object initializers can now use indexers, but there are trade-offs between using indexers and collection initializers.
- The null conditional `? .` operator makes it much easier to work with chained operations in which one element of the chain can return `null`.
- Exception filters allow more control over exactly which exceptions are caught based on the exception's data rather than just its type.

Part 4

C# 7 and beyond

C

7 is the first release since C# 1 to have multiple minor releases.¹ There have been four releases:

- C# 7.0 in March 2017 with Visual Studio 2017 version 15.0
- C# 7.1 in August 2017 with Visual Studio 2017 version 15.3
- C# 7.2 in December 2017 with Visual Studio 2017 version 15.5
- C# 7.3 in May 2018 with Visual Studio 2017 version 15.7

Most of the minor releases have expanded on new features introduced in earlier C# 7.x releases rather than introducing entirely new areas, although the ref-related features covered in chapter 13 were greatly expanded in C# 7.2.

As far as I'm aware, no plans exist for a C# 7.4 release, although I wouldn't completely rule it out. Having multiple versions seems to have worked reasonably well, and I expect the same sort of release cycle for C# 8.

There's more to talk about in C# 7 than in C# 6, because the features are more complex. Tuples have an interesting separation between the types as the compiler considers them and the types that the CLR uses. Local methods fascinate me in terms of comparing their implementation with that of lambda expressions. Pattern matching is reasonably simple to understand but requires a certain amount of thought in terms of using it to its best advantage. The ref-related features are inherently complicated even when they sound simple. (I'm looking at you, `in` parameters.)

¹ Visual Studio 2002 included C# 1.0, and Visual Studio 2003 included C# 1.2. I've no idea why the version number skipped 1.1, and it's not clear what the differences were between the two versions.

Although I expect most developers to find most C# 6 features useful every day, you may find some of the C# 7 features aren't useful to you at all. I rarely use tuples in my code, because I usually target platforms where they're not available. I don't use the ref-related features much, as I'm not coding in a context where they're particularly useful. This doesn't stop them from being good features; they're just not universally applicable. Other C# 7 features, such as pattern matching, throw expressions, and numeric literal improvements, are more likely to be useful to all developers but perhaps with less impact than the more targeted features.

I mention all of this merely to set expectations. As always, when you read about a feature, consider how you might apply it in your own code. Don't feel forced to apply it; there are no points for using the most language features in the shortest amount of code. If you find you don't have a use for that feature right now, that's fine. Just remember it's there so if you're in a different context later, you know what's available.

It's also important for me to set expectations about chapter 15, which looks at the future of C#. Most of the chapter demonstrates features already available in C# 8 preview builds, but there's no guarantee that all of those features will ship in the final build, and there may well be other features I haven't mentioned at all. I hope you will find the features I've written about as exciting as I do and will keep watch for new previews and blog posts by the C# team. This is an exciting time to be a C# developer, both in terms of what we have today and the promise of a bright future.

11

Composition using tuples

This chapter covers

- Using tuples to compose data
- Tuple syntax: literals and types
- Converting tuples
- How tuples are represented in the CLR
- Alternatives to tuples and guidelines for their use

Back in C# 3, LINQ revolutionized how we write code to handle collections of data. One of the ways it did that was to allow us to express many operations in terms of how we want to handle each individual item: how to transform an item from one representation to another, or how to filter items out of the result, or how to sort the collection based on a particular aspect of each item. For all that, LINQ didn't give us many new tools for working with noncollections.

Anonymous types provide one kind of composition but with the huge restriction of being useful only within a block of code. You can't declare that a method returns an anonymous type precisely because the return type can't be named.

C# 7 introduces support for tuples to make data composition simple along with deconstruction from a composite type into its individual components. If you're

now thinking to yourself that C# already has tuples in the form of the `System.Tuple` types, you're right to an extent; those types already exist in the framework but don't have any language support. To add to the confusion, C# 7 doesn't use those tuple types for its language-supported tuples. It uses a new set of `System.ValueTuple` types, which you'll explore in section 11.4. There's a comparison with `System.Tuple` in section 11.5.1.

11.1 Introduction to tuples

Tuples allow you to create a single composite value from multiple individual values. They're shorthand for composition with no extra encapsulation for situations where values are related to each other but you don't want the work of creating a new type. C# 7 introduces new syntax to make working with tuples simple.

As an example, suppose you have a sequence of integers and you want to find both the minimum and the maximum in one pass. This sounds like you should be able to put that code into a single method, but what would you make the return type? You could return the minimum value and use an `out` parameter for the maximum value or use two `out` parameters, but both of those feel fairly clunky. You could create a separate named type, but that's a lot of work for just one example. You could return a `Tuple<int, int>` using the `Tuple<, >` class introduced in .NET 4, but then you couldn't easily tell which was the minimum value and which was the maximum (and you'd end up allocating an object just to return the two values). Or you could use C# 7's tuples. You could declare the method like this:

```
static (int min, int max) MinMax(IEnumerable<int> source)
```

Then you could call it like this:

```
int[] values = { 2, 7, 3, -5, 1, 0, 10 };
var extremes = MinMax(values);
Console.WriteLine(extremes.min);
Console.WriteLine(extremes.max);
```

The diagram illustrates the flow of data. An arrow points from the variable `values` to the method call `MinMax(values)`. A bracket labeled "Calls the method to compute the min and max and returns them as a tuple" is placed over the method call. Another arrow points from the method call to the variable `extremes`. A bracket labeled "Prints out the minimum value (-5)" is placed over the line `Console.WriteLine(extremes.min)`. A final arrow points from the variable `extremes` to the line `Console.WriteLine(extremes.max)`. A bracket labeled "Prints out the maximum value (10)" is placed over the line `Console.WriteLine(extremes.max)`.

You'll look at a couple of implementations of `MinMax` shortly, but this example should give you enough of an idea about the purpose of the feature to make it worth reading all the fairly detailed descriptions over the course of the chapter. For a feature that sounds simple, there's quite a lot to say about tuples, and it's all interrelated, which makes it hard to describe in a logical order. If you find yourself asking "But what about...?" while reading, I urge you to mentally put a pin in the question until the end of the section. Nothing here is complex, but there's a lot to get through, especially because I'm aiming to be comprehensive. Hopefully, by the time you reach the end of the chapter, all your questions will be answered.¹

¹ If they're not, you should ask for more information on the Author Online forum or Stack Overflow, of course.

11.2 Tuple literals and tuple types

You can think of tuples as the introduction of some types into the CLR and some syntactic sugar to make those types easy to use, both in terms of specifying them (for variables and so on) and constructing values. I'm going to start off explaining everything from the perspective of the C# language without worrying too much about how it maps onto the CLR; then I'll loop back to explain everything the compiler is doing for you behind the scenes.

11.2.1 Syntax

C# 7 introduces two new pieces of syntax: tuple literals and tuple types. They look similar: they're both comma-separated sequences of two or more elements in parentheses. In a *tuple literal*, each element has a value and an optional name. In a *tuple type*, each element has a type and an optional name. Figure 11.1 shows an example of a tuple literal; figure 11.2 shows an example of a tuple type. Each has one named element and one unnamed element.

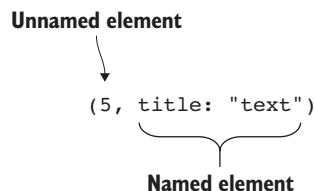


Figure 11.1 A tuple literal with element values 5 and "text". The second element is named title.

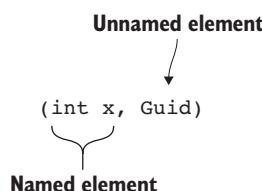


Figure 11.2 A tuple type with element types int and Guid. The first element is named x.

In practice, it's more common for either all the elements to be named or none of them to be named. For example, you might have tuple types of (int, int) or (int x, int y, int z), and you might have tuple literals of (x: 1, y: 2) or (1, 2, 3). But this is a coincidence; nothing is tying the elements together in terms of whether they have names. There are two restrictions on names to be aware of, though:

- The names have to be unique within the type or literal. A tuple literal of (x: 1, x: 2) isn't allowed and wouldn't make any sense.
- Names of the form ItemN, where N is an integer, are allowed only where the value of N matches the position in the literal or type, starting at 1. So (Item1: 0, Item2: 0) is fine, but (Item2: 0, Item1: 0) is prohibited. You'll see why this is the case in the next section.

Tuple types are used to specify types in the same places other type names are used: variable declarations, method return types, and so on. Tuple literals are used like any other expression specifying a value; they simply compose those elements into a tuple value.

The element values in a tuple literal can be any value other than a pointer. Most of the examples in this chapter use constants (primarily integers and strings) for convenience, but you'll often use variables as the element values in a literal. Similarly, the element types in a tuple can be any nonpointer type: arrays, type parameters, even other tuple types.

Now that you know what a tuple type looks like, you can understand the return type (`int min, int max`) for our `MinMax` method:

- It's a tuple type with two elements.
- The first element is an `int` named `min`.
- The second element is an `int` named `max`.

You also know how to create a tuple by using a tuple literal, so you can implement our method completely, as shown in the following listing.

Listing 11.1 Representing the minimum and maximum values of a sequence as a tuple

```
static (int min, int max) MinMax(           ← Return type is a tuple
    IEnumerable<int> source)                  with named elements.
{
    using (var iterator = source.GetEnumerator())
    {
        if (!iterator.MoveNext())           ← Prohibits empty
        {                                    sequences
            throw new InvalidOperationException(
                "Cannot find min/max of an empty sequence");
        }
        int min = iterator.Current;          | Uses regular int variables
        int max = iterator.Current;          | to keep track of min/max
        while (iterator.MoveNext())
        {
            min = Math.Min(min, iterator.Current);   | Updates the variables
            max = Math.Max(max, iterator.Current);   | with the new min/max
        }
        return (min, max);                  ← Constructs a tuple
    }                                    from min and max
}
```

The only parts of listing 11.1 that involve new features are the return type that I've already explained and the `return` statement, which uses a tuple literal:

```
return (min, max);
```

So far, I haven't talked about the type of a tuple literal. I've only said that they're used to create tuple values, but I'm going to deliberately leave that somewhat vague at the moment. I'll note that our tuple literal doesn't have any element names at the moment, at least not in C# 7.0. The `min` and `max` parts provide the values for the elements using the local variables in the method.

Good tuple element names match good variable names

Is it a coincidence that the variable names used in the literal match the names used in method's return type? As far as the compiler is concerned, absolutely. The compiler wouldn't care if you declared the method to return `(waffle: int, iceCream : int)`.

For a human reader, it's far from a coincidence; the names indicate that the values have the same meaning in the returned tuple as they do within the method. If you find yourself providing very different names, you might want to check whether you have a bug or whether perhaps some of the names could be chosen better.

While we're defining terms, let's define the *arity* of a tuple type or literal as the number of elements it has. For example, `(int, long)` has an arity of 2, and `("a", "b", "c")` has an arity of 3. The element types themselves are irrelevant to the arity.

NOTE This isn't new terminology, really. The concept of arity already exists in generics, where the arity is the number of type parameters. The `List<T>` type has an arity of 1, whereas `Dictionary< TKey, TValue >` has an arity of 2.

The tip around good element names matching good variable names really gives a hint as to an aspect of tuple literals that was improved in C# 7.1.

11.2.2 Inferred element names for tuple literals (C# 7.1)

In C# 7.0, tuple element names had to be explicitly stated in code. This would often lead to code that looked redundant: the names specified in the tuple literal would match the property or local variable names used to provide the values. In the simplest form, this might be something like the following:

```
var result = (min: min, max: max);
```

The inference doesn't just apply when your code uses simple variables, though; tuples are often initialized from properties, too. This is particularly prevalent in LINQ with projections.

In C# 7.1, tuple element names are inferred when the value comes from a variable or property in exactly the same way as names are inferred in anonymous types. To see how useful this is, let's consider three ways of writing a query in LINQ to Objects that joins two collections to obtain the names, job titles, and departments of employees. First, here's traditional LINQ using anonymous types:

```
from emp in employees
join dept in departments on emp.DepartmentId equals dept.Id
select new { emp.Name, emp.Title, DepartmentName = dept.Name };
```

Next, we'll use tuples with explicit element names:

```
from emp in employees
join dept in departments on emp.DepartmentId equals dept.Id
select (name: emp.Name, title: emp.Title, departmentName: dept.Name);
```

Finally, we'll use inferred element names with C# 7.1:

```
from emp in employees
join dept in departments on emp.DepartmentId equals dept.Id
select (emp.Name, emp.Title, DepartmentName: dept.Name);
```

This changes the case of the tuple elements compared with the previous example but still achieves the goal of creating tuples with useful names using concise code.

Although I've demonstrated the feature within a LINQ query, it applies anywhere you use tuple literals. For example, given a list of elements, you could create a tuple with the count, min, and max by using element name inference for the count:

```
List<int> list = new List<int> { 5, 1, -6, 2 };
var tuple = (list.Count, Min: list.Min(), Max: list.Max());
Console.WriteLine(tuple.Count);
Console.WriteLine(tuple.Min);
Console.WriteLine(tuple.Max);
```

Note that you still need to specify element names for `Min` and `Max`, because those values are obtained using method invocations. Method invocations don't provide inferred names for either tuple elements or anonymous type properties.

As one slight wrinkle, if two names would both be inferred to be the same, neither is inferred. If there's a collision between an inferred name and an explicit name, the explicit name takes priority and the other element remains unnamed. Now that you know how to specify tuple types and tuple literals, what can you do with them?

11.2.3 Tuples as bags of variables

The next sentence may come as a shock to you, so please prepare yourself: tuple types are value types with public, read/write fields. Surely not! I usually recommend against mutable value types in the strongest possible terms, and likewise I always suggest that fields should be private. In general, I stand by those recommendations, but tuples are slightly different.

Most types aren't just raw data; they attach meaning to that data. Sometimes there's validation for the data. Sometimes there's an enforced relationship between multiple pieces of data. Usually, there are operations that make sense only because of the meaning attached to the data.

Tuples don't do that at all. They just act as if they were bags of variables. If you have two variables, you can change them independently; there's no inherent connection between them, and there's no enforced relationship between them. Tuples allow you to do exactly the same thing, but with the extra feature that you can pass that whole bag of variables around in one value. This is particularly important when it comes to methods, which can return only a single value.

Figure 11.3 shows this graphically. The left side shows code and a mental model for declaring three independent local variables, and the right side shows similar code, but two of those variables are in a tuple (the oval). On the right side, the name and score are grouped together as a tuple in the player variable. When you want to treat them

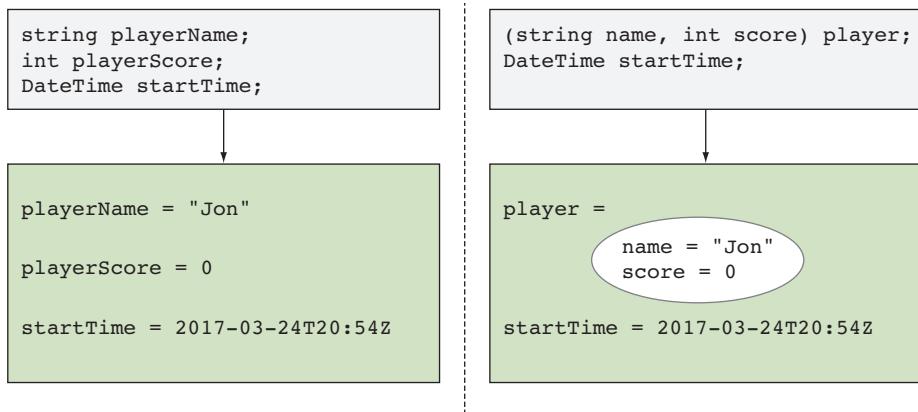


Figure 11.3 Three separate variables on the left; two variables, one of which is a tuple, on the right

as separate variables, you can still do so (for example, printing out `player.score`), but you can also treat them as a group (for example, assigning a new value to `player`).

Once you get into the mentality of thinking of a tuple as a bag of variables, a lot of things start to make more sense. But what are those variables? You've already seen that when you have named elements in a tuple you can refer to them by name, but what if an element doesn't have a name?

ACCESSING ELEMENTS BY NAME AND POSITION

You may recall that there's a restriction on element names of the form `ItemN`, where `N` is a number. Well, that's because every variable in a tuple can be referred to by its position as well as by any name it was given. There's still only one variable per element; it's just that there may be two ways of referring to that variable. It's easiest to show this with an example as in the following listing.

Listing 11.2 Reading and writing tuple elements by name and position

```
var tuple = (x: 5, 10);
Console.WriteLine(tuple.x);
Console.WriteLine(tuple.Item1);
Console.WriteLine(tuple.Item2);
```

Displays the first element by name and position

```
tuple.x = 100;
Console.WriteLine(tuple.Item1);
```

Modifies the first element by name

Displays the first element by position (prints 100)

The second element has no name; can use only position.

At this point, you can probably see why `(Item1: 10, 20)` is okay but `(Item2: 10, 20)` isn't allowed. In the first case, you're redundantly naming the element, but in the second case, you're causing ambiguity as to whether `Item2` refers to the first element

(by name) or the second element (by position). You could argue that `(Item5: 10, 20)` should be allowed because there are only two elements; `Item5` doesn't exist because the tuple has only two elements. This is one of those cases where even though something wouldn't technically cause an ambiguity, it'd certainly be confusing, so it's still prohibited.

Now that you know you can modify a tuple value after creating it, you can rewrite your `MinMax` method to use a single tuple local variable for the “result so far” instead of your separate `min` and `max` variables, as shown in the next listing.

Listing 11.3 Using a tuple instead of two local variables in `MinMax`

```
static (int min, int max) MinMax(IEnumerable<int> source)
{
    using (var iterator = source.GetEnumerator())
    {
        if (!iterator.MoveNext())
        {
            throw new InvalidOperationException(
                "Cannot find min/max of an empty sequence");
        }
        var result = (min: iterator.Current,
                     max: iterator.Current); | Constructs a tuple with the
                                         first value as both min and max
        while (iterator.MoveNext())
        {
            result.min = Math.Min(result.min, iterator.Current);
            result.max = Math.Max(result.max, iterator.Current);
        }
        return result;           ← Returns the
    }                         tuple directly
}
```

Modifies each field of the tuple separately

Listing 11.3 is very, very close to listing 11.1 in terms of how it works. You've just grouped two of your four local variables together; instead of `source`, `iterator`, `min`, and `max`, you have `source`, `iterator`, and `result`, where `result` has `min` and `max` elements inside it. The memory usage will be the same and the performance will be the same; it's just a different way of writing it. Is it a better way of writing the code? That's fairly subjective, but at least it's a localized decision; it's purely an implementation detail.

TREATING A TUPLE AS A SINGLE VALUE

While you're thinking about alternative implementations for your method, let's consider another one. You can take this code that first assigns a new value to `result.min` and then a new value to `result.max`:

```
result.min = Math.Min(result.min, iterator.Current);
result.max = Math.Max(result.max, iterator.Current);
```

If you assign directly to `result` instead, you can replace the whole bag in a single assignment, as shown in the following listing.

Listing 11.4 Reassigning the result tuple in one statement in MinMax

```

static (int min, int max) MinMax(IEnumerable<int> source)
{
    using (var iterator = source.GetEnumerator())
    {
        if (!iterator.MoveNext())
        {
            throw new InvalidOperationException(
                "Cannot find min/max of an empty sequence");
        }
        var result = (min: iterator.Current, max: iterator.Current);
        while (iterator.MoveNext())
        {
            result = (Math.Min(result.min, iterator.Current),
                      Math.Max(result.max, iterator.Current));
        }
        return result;
    }
}

```

Assigns a new value to the whole of result

Again, there's not an awful lot to choose between implementations, and that's because in listing 11.3 the two elements of the tuple were being updated individually, referring only to the previous value of the same element. A more compelling example is to write a method that returns the Fibonacci sequence² as an `IEnumerable<int>`. C# already helps you do that by providing iterators with `yield`, but it can be a bit fiddly. The following listing shows a perfectly reasonable C# 6 implementation.

Listing 11.5 Implementing the Fibonacci sequence without tuples

```

static IEnumerable<int> Fibonacci()
{
    int current = 0;
    int next = 1;
    while (true)
    {
        yield return current;
        int nextNext = current + next;
        current = next;
        next = nextNext;
    }
}

```

As you iterate, you keep track of the current element and the next element of the sequence. In each iteration, you shift from the pair representing “current and next” to “next and next-next.” To do that, you need a temporary variable; you can't simply assign new values directly to `current` and `next` one after the other, because the first assignment would lose information you need for the second assignment.

² The first two elements are 0 and 1; after that, any element of the sequence is the sum of the previous two elements.

Tuples let you perform a single assignment that changes both elements. The temporary variable is still present in the IL, but the resulting source code shown in the following listing ends up being beautiful, in my view.

Listing 11.6 Implementing the Fibonacci sequence with tuples

```
static IEnumerable<int> Fibonacci()
{
    var pair = (current: 0, next: 1);
    while (true)
    {
        yield return pair.current;
        pair = (pair.next, pair.current + pair.next);
    }
}
```

After you've gone that far, it's hard to resist generalizing this further to generate arbitrary sequences, extracting all of the Fibonacci code out to just arguments in a method call. The following listing introduces a generalized `GenerateSequence` method suitable for generating all kinds of sequences based on its arguments.

Listing 11.7 Separating concerns of sequence generation for Fibonacci

```
static IEnumerable<TResult>
GenerateSequence<TState, TResult>(
    TState seed,
    Func<TState, TState> generator,
    Func<TState, TResult> resultSelector)
{
    var state = seed;
    while (true)
    {
        yield return resultSelector(state);
        state = generator(state);
    }
}
```

Method to allow the generation of arbitrary sequences based on previous state

Sample usage

```
var fibonacci = GenerateSequence(
    (current: 0, next: 1),
    pair => (pair.next, pair.current + pair.next),
    pair => pair.current);
```

Use of sequence generator specifically for the Fibonacci sequence

This could certainly be achieved using anonymous or even named types, but it wouldn't be as elegant. Readers with experience in other programming languages may not be overly impressed by this—it's not as if C# 7 has brought a brand-new paradigm to the world—but it's exciting to be able to write code as beautiful as this in C#.

Now that you've seen the basics of how tuples work, let's dive a bit deeper. In the next section, we'll mostly be considering conversions, but we'll also look at where element names are important and where they're not.

11.3 Tuple types and conversions

Until now, I've carefully avoided going into the details of the type of a tuple literal. By staying somewhat vague, I've been able to show quite a lot of code so you can get a feeling for how tuples can be used. Now's the time to justify the *In Depth* part of the book's title. First, think about all the declarations you've seen using `var` and tuple literals.

11.3.1 Types of tuple literals

Some tuple literals have a type, but some don't. It's a simple rule: a tuple literal has a type when every element expression within it has a type. The idea of an expression without a type is nothing new in C#: lambda expressions, method groups, and the `null` literal are also expressions with no type. Just as in those examples, you can't use tuple literals without a type to assign a value to an implicitly typed local variable. For example, this is valid, because both 10 and 20 are expressions with a type:

```
var valid = (10, 20);
```

But this is invalid because the `null` literal doesn't have a type:

```
var invalid = (10, null);
```

Just like the `null` literal, a tuple literal without a type can still be convertible to a type. When a tuple has a type, any element names are also part of the type.

For example, in each of these cases, the left side is equivalent to the right side:

<code>var tuple = (x: 10, 20);</code>	<code>(int x, int) tuple = (x: 10, 20);</code>
<code>var array = new[] {("a", 10)};</code>	<code>(string, int) [] array = {("a", 10)};</code>
<code>string[] input = {"a", "b" };</code>	<code>string[] input = {"a", "b" };</code>
<code>var query = input</code> <code>.Select(x => (x, x.Length));</code>	<code>IEnumerable<(string, int)> query =</code> <code>input.Select<string, (string, int)></code> <code>(x => (x, x.Length));</code>

The first example demonstrates how element names propagate from tuple literals to tuple types. The last example shows how the type inference still works in complex ways: the type of `input` allows the type of `x` in the lambda expression to be fixed to `string`, which then allows the expression `x.Length` to be bound appropriately. This leaves a tuple literal with element types `string` and `int`, so the return type of the lambda expression is inferred to be `(string, int)`. You saw a similar kind of inference in listing 11.7 with our Fibonacci implementation using the sequence generator method, but you weren't focusing on the types involved at the time.

That's fine for tuple literals that have types. But what can you do with tuple literals that don't have types? How can you convert from a tuple literal without names to a tuple type with names? To answer these questions, you need to look at tuple conversions in general.

You need to think about two kinds of conversions: conversions from tuple literals to tuple types and conversions from one tuple type to another. You've already seen this kind of difference in chapter 8: there's a conversion from an interpolated string literal expression to `FormattableString` but no conversion from the `string` type to `FormattableString`. The same idea is at work here. You'll look first at the literal conversions.

Lambda expression parameters can look like tuples

Lambda expressions with a single parameter aren't confusing, but if you use two parameters, they can look like tuples. As an example, let's look at a useful method that just uses the LINQ `Select` overload that provides the projection with the index of the element as well as the value. It's often useful to propagate the index through the other operations, so it makes sense to put the two pieces of data in a tuple. That means you end up with this method:

```
static IEnumerable<(T value, int index)> WithIndex<T>
    (this IEnumerable<T> source) =>
    source.Select((value, index) => (value, index));
```

Concentrate on the lambda expression:

```
(value, index) => (value, index)
```

Here the first occurrence of `(value, index)` isn't a tuple literal; it's the sequence of parameters for the lambda expression. The second occurrence *is* a tuple literal, the result of the lambda expression.

There's nothing wrong here. I just don't want it to take you by surprise when you see something similar.

11.3.2 Conversions from tuple literals to tuple types

Just as in many other parts of C#, there are implicit conversions from tuple literals and explicit conversions. I expect the use of explicit conversions to be rare for reasons I'll show in a moment. But after you understand how implicit conversions work, the explicit conversions pretty much fall out anyway.

IMPLICIT CONVERSIONS

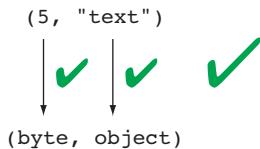
A tuple literal can be implicitly converted to a tuple type if both of the following are true:

- The literal and the type have the same arity.
- Each expression in the literal can be implicitly converted to its corresponding element type.

The first bullet is simple. It'd be odd to be able to convert `(5, 5)` to `(int, int, int)`, for example. Where would the last value come from? The second bullet is a little more complex, but I'll clarify it with examples. First, let's try this conversion:

```
(byte, object) tuple = (5, "text");
```

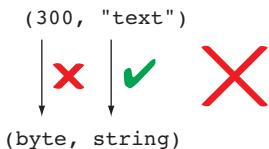
As per the preceding description, you need to look at each element expression in the source tuple literal `(5, "text")` and check whether there's an implicit conversion to the corresponding element type in the target tuple type `(byte, object)`. If every element can be converted, the conversion is valid:



Even though there's no implicit conversion from `int` to `byte`, there's an implicit conversion from the integer constant `5` to `byte` (because `5` is in the range of valid `byte` values). There's also an implicit conversion from a string literal to `object`. All the conversions are valid, so the whole conversion is valid. Hooray! Now let's try a different conversion:

```
(byte, string) tuple = (300, "text");
```

Again, you try to apply implicit conversions element-wise:



In this case, you're trying to convert the integer constant `300` to `byte`. That's outside the range of valid values, so there's no implicit conversion. There's an explicit conversion, but that doesn't help when you're trying to achieve an overall implicit conversion of the tuple literal. There's an implicit conversion from the string literal to the `string` type, but because not all the conversions are valid, the whole conversion is invalid. If you try to compile this code, you'll get an error pointing to the `300` within the tuple literal:

```
error CS0029: Cannot implicitly convert type 'int' to 'byte'
```

This error message is a little misleading. It suggests that our previous example shouldn't be valid either. The compiler isn't really trying to convert the type `int` to `byte`; it's trying to convert the expression `300` to `byte`.

EXPLICIT CONVERSIONS

Explicit conversions for tuple literals follow the same rules as implicit conversions, but they require an explicit conversion to be present for each element expression to the corresponding type. If that condition is met, there's an explicit conversion from the tuple literal to the tuple type, so you can cast in the normal way.

TIP Every implicit conversion in C# also counts as an explicit conversion, which is somewhat confusing. You can think of the condition as “there has to be a conversion, either explicit or implicit, available for each element” if you find that clearer.

To go back to our conversion of `(300, "text")`, there’s an explicit conversion to the tuple type `(byte, string)`. But converting that exact expression requires an unchecked context for the conversion to work, because the compiler knows that the constant value 300 is outside the normal range of `byte`. A more realistic example would use an `int` variable from elsewhere:

```
int x = 300;
var tuple = ((byte, string)) (x, "text");
```

The casting part—`((byte, string))`—looks like it has more parentheses than it needs, but they’re all required. The inner ones are specifying the tuple type, and the outer ones are signifying the cast. Figure 11.4 shows this graphically.

It looks ugly to me, but it’s nice that it’s at least available. A simpler alternative in many cases is to write the appropriate cast in each element expression in the tuple literal, at which point not only would the tuple conversion be valid, but the inferred type of the literal becomes what you want anyway. For example, I’d probably write the preceding example as follows:

```
int x = 300;
var tuple = ((byte) x, "text");
```

The two options are equivalent; when the conversion is applied to the whole tuple literal, the compiler still emits an explicit conversion for each element expression. But I find the latter much more readable. Aside from anything else, it shows clearer intent: you know an explicit conversion is required from `int` to `byte`, but you’re happy for the `string` to stay as it is. If you were trying to convert several values to a specific tuple type (rather than using the inferred type), this would help to make it clear which conversions are explicit and therefore potentially lossy instead of accidentally losing data due to a whole-tuple explicit conversion.

THE ROLE OF ELEMENT NAMES IN TUPLE LITERAL CONVERSIONS

You may have noticed that this section hasn’t mentioned names at all. They’re almost entirely irrelevant within tuple literal conversions. Most important, it’s fine to convert from an element expression without a name to a type element with a name. You’ve been doing that a lot in this chapter without me raising it as an issue. You did it right from the start with our first `MinMax` method implementation. As a reminder, the method was declared as follows:

```
static (int min, int max) MinMax(IEnumerable<int> source)
```

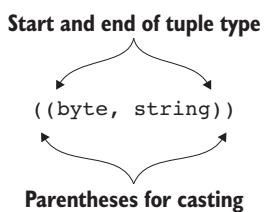


Figure 11.4 Explaining the parentheses in an explicit tuple conversion

And then our return statement was this:

```
return (min, max);
```

You're trying to convert a tuple literal with no element names³ to `(int min, int max)`. Of course, it's valid; otherwise, I wouldn't have shown it to you. It's also convenient. Element names aren't completely irrelevant in tuple literal conversions, though. When an element name is explicitly specified in the tuple literal, the compiler will warn you if either there's no corresponding element name in the type you're converting it to or the two names are different. Here's an example:

```
(int a, int b, int c, int, int) tuple =
    (a: 10, wrong: 20, 30, pointless: 40, 50);
```

This shows all the possible combinations for element names in this order:

- 1** Both the target type and the tuple literal specify the same element name.
- 2** Both the target type and the tuple literal specify a name for the element, but the names are different.
- 3** The target type specifies an element name, but the tuple literal doesn't.
- 4** The target type doesn't specify an element name, but the tuple literal does.
- 5** Neither the target type nor the tuple literal specifies an element name.

Of these, the second and the fourth result in compile-time warnings. The result of compiling that code is shown here:

```
warning CS8123: The tuple element name 'wrong' is ignored because a different
name is specified by the target type '(int a, int b, int c, int, int)'.
warning CS8123: The tuple element name 'pointless' is ignored because a
different name is specified by the target type '(int a, int b, int c,
int, int)'
```

The second warning message isn't as helpful as it might be, because the target type isn't specifying a name at all for the corresponding element. Hopefully, you could still work out what's wrong.

Is this useful? Absolutely. Not when you're declaring a variable and constructing a value in one statement, but when the declaration and the construction are separated. For example, suppose our `MinMax` method in listing 11.1 had been really long in a way that was hard to refactor. Should you return `(min, max)` or `(max, min)`? Yes, in this case just the name of the method makes the order pretty obvious, but in some cases, it might not be as clear. At that point, adding element names to the `return` statement can be used as validation. This compiles warning free:

```
return (min: min, max: max);
```

But if you reverse the elements, you get a warning for each element:

```
return (max: max, min: min);           ←———— Warning CS8123, twice
```

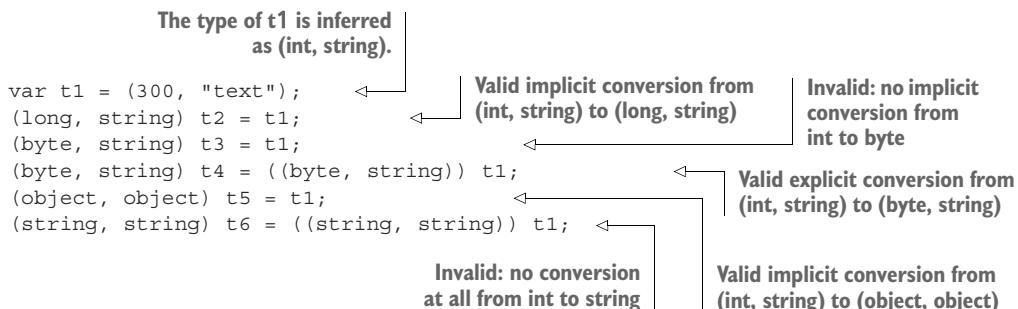
³ In C# 7.0, at least. As noted in section 11.2.2, in C# 7.1 the names are inferred.

Note that this applies only to explicitly specified names. Even in C# 7.1, when element names are inferred from a tuple literal of `(max, min)`, that doesn't generate a warning when you convert it to a tuple type of `(int min, int max)`.

I always prefer to structure the code to make this so clear that you don't need this extra checking. But it's good to know that it's available when you need it, perhaps as a first step before you refactor the method to be shorter, for example.

11.3.3 Conversions between tuple types

After you have the hang of tuple literal conversions, implicit and explicit tuple type conversions are reasonably simple, because they work in a similar way. Here, you have no expressions to worry about, just the types. There's an implicit conversion from a source tuple type to a target tuple type of the same arity if there's an implicit conversion from each source element type to the corresponding target element type. Similarly, there's an explicit conversion from a source tuple type to a target tuple type of the same arity if there's an explicit conversion from each source element type to the corresponding target element type. Here's an example showing multiple conversions all from a source type of `(int, string)`:



In this case, the explicit conversion from `(int, string)` to `(byte, string)` in the fourth line will result in the value of `t4.Item1` being 44, because that's the result of the explicit conversion of the `int` value 300 to `byte`.

Unlike with tuple literal conversions, there's no warning if element names don't match up. I can show this with an example that's similar to our arity-5 conversion with tuple literals. All you need to do is store the tuple value in a variable first so that you perform a type-to-type conversion instead of a literal-to-type conversion:

```
var source = (a: 10, wrong: 20, 30, pointless: 40, 50);
(int a, int b, int c, int, int) tuple = source;
```

This compiles with no warnings at all. One aspect of tuple type conversion is important in a way that isn't applicable for literal conversions, however, and that's when the conversion isn't just an implicit conversion but is an identity conversion.

TUPLE TYPE IDENTITY CONVERSIONS

The concept of identity conversions has been present in C# since the beginning, although it's been expanded over time. Before C# 7, the rules worked like this:

- An identity conversion exists from each type to itself.
- An identity conversion exists between `object` and `dynamic`.
- An identity conversion exists between two array types if an identity conversion exists between their element types. For example, an identity conversion exists between `object[]` and `dynamic[]`.
- Identity conversions extend to constructed generic types when identity conversions exist between corresponding type arguments. For example, an identity conversion exists between `List<object>` and `List<dynamic>`.

Tuples introduce another kind of identity conversion: between tuple types of the same arity when an identity conversion exists between each corresponding pair of element types, regardless of name. In other words, identity conversions exist (in both directions; identity conversions are always symmetric) between the following types:

- `(int x, object y)`
- `(int a, dynamic d)`
- `(int, object)`

Again, this can be applied to constructed types, and the tuple element types can be constructed, too, so long as an identity conversion is still available. So, for example, identity conversions exist between these two types:

- `Dictionary<string, (int, List<object>) >`
- `Dictionary<string, (int index, List<dynamic> values) >`

Identity conversions are mostly important for tuples when it comes to constructed types. It'd be annoying if you could easily convert from `(int, int)` to `(int x, int y)` but not from `IEnumerable<(int, int)>` to `IEnumerable<(int x, int y)>`, or vice versa.

The identity conversions are important for overloads as well. In the same way two overloads can't vary just by return type, they can't vary only by parameter types with identity conversions between them. You can't write two methods in the same class like this:

```
public void Method((int, int) tuple) {}  
public void Method((int x, int y) tuple) {}
```

If you do so, you'll receive a compile-time error like this:

```
error CS0111: Type 'Program' already defines a member called 'Method' with  
the same parameter types
```

From a C# language perspective, the parameter types aren't exactly the same, but making the error message absolutely precise in terms of identity conversions would make it a lot harder to understand.

If you find the official definitions of identity conversions confusing, one simple (though rather less official) way of thinking about them is this: two types are identical if you can't tell the difference between them at execution time. We'll go into a lot more detail about that in section 11.4.

LACK OF GENERIC VARIANCE CONVERSIONS

With the identity conversions in mind, you might be hopeful that you could use tuple types with generic variance for interface and delegate types. Sadly, this isn't the case. Variance applies only to reference types, and tuple types are always value types. As an example, it feels like this should compile:

```
IEnumerable<(string, string)> stringPairs = new (string, string)[10];
IEnumerable<(object, object)> objectPairs = stringPairs;
```

But it doesn't. Sorry about that. I can't see it coming up terribly often as a practical issue, but I wanted to remove the disappointment you might feel if you ever wanted this and expected it to work.

11.3.4 Uses of conversions

Now that you know what's available, you may be wondering when you'd want to use these tuple conversions. This will largely depend on how you use tuples in a broader sense. Tuples used within a single method or returned from private methods to be used in the same class are rarely going to require conversions. You'll just pick the right type to start with, possibly casting within a tuple literal when constructing an initial value.

It's more likely that you'll need to convert from one tuple type to another when you're using internal or public methods accepting or returning tuples, because you'll have less control over the element types. The more broadly a tuple type is used, the less likely it is to be *exactly* the desired type in every single use.

11.3.5 Element name checking in inheritance

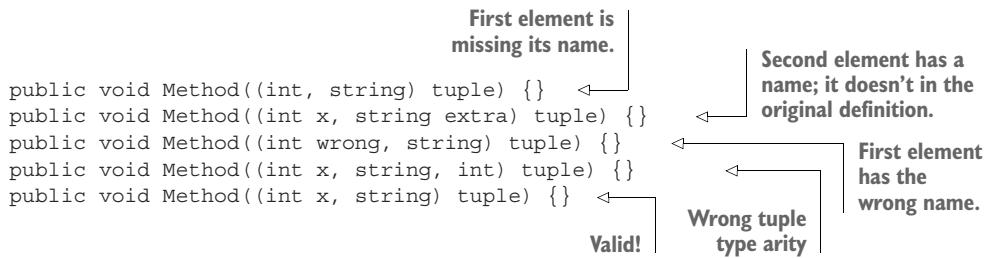
Although element names aren't important in conversions, the compiler is picky about their use in inheritance. When a tuple type appears in a member you're either overriding from a base class or implementing from an interface, the element names you specify must match those in the original definition. Not only must any names that are specified in the original definition be matched, but if there isn't a name in the original definition, you can't put one in the implementation. The element types in the implementation have to be identity convertible to the element types in the original definition.

As an example, consider this `ISample` interface and some methods trying to implement `ISample.Method` (each of which would be in a separate implementation class, of course):

```
interface ISample
{
    void Method((int x, string) tuple);
}

public void Method((string x, object) tuple) {}
```





That example deals only with an interface implementation, but the same restrictions hold when overriding a base class member. Likewise, that example uses only a parameter, but the restrictions apply to return types, too. Note that this means that adding, removing, or changing a tuple element name in an interface member or a virtual/abstract class member is a breaking change. Think carefully before doing this in a public API!

NOTE In some senses, this is a slightly inconsistent step, in that the compiler has never worried before about the author of a class changing method parameter names when overriding a method or implementing an interface. The ability to specify argument names means that this can cause problems if a caller changes their code in terms of whether they refer to the interface or the implementation. My suspicion is that if the C# language designers were starting again from scratch, this would be prohibited, too.

C# 7.3 has added one more language feature to tuples: the ability to compare them with `==` and `!=` operators.

11.3.6 Equality and inequality operators (C# 7.3)

As you'll see in section 11.4.5, the CLR representation of value tuples has supported equality via the `Equals` method from the start. But it doesn't overload the `==` or `!=` operators. As of C# 7.3, however, the compiler provides `==` and `!=` implementations between tuples where there's an identity conversion between the tuple types of the two operands. (Aside from other aspects of identity conversions, that means the element names aren't important.)

The compiler expands the `==` operator into element-wise comparisons with the `==` operators of each corresponding pair of values and the `!=` operator into element-wise comparisons with the `!=` operators of each corresponding pair of values. That's probably easiest to show with the following example.

Listing 11.8 Equality and inequality operators

```
var t1 = (x: "x", y: "y", z: 1);
var t2 = ("x", "y", 1);
Console.WriteLine(t1 == t2);
```

← **Equality operator**

```

Console.WriteLine(t1.Item1 == t2.Item1 &&
                  t1.Item2 == t2.Item2 &&
                  t1.Item3 == t2.Item3);

```

Equivalent code
generated by compiler

```

Console.WriteLine(t1 != t2);           ← Inequality operator
Console.WriteLine(t1.Item1 != t2.Item1 &&
                  t1.Item2 != t2.Item2 &&
                  t1.Item3 != t2.Item3);

```

Equivalent code
generated by compiler

Listing 11.8 shows two tuples (one with element names and one without) and compares them for equality and inequality. In each case, I've then shown what the compiler generates for that operator. The important point to note here is that the generated code uses any overloaded operators provided by the element types. It'd be impossible for the CLR type to provide the same functionality without using reflection. This is a task better handled by the compiler.

We've now gone as far into the language rules of tuples as we need to. The precise details of how element names are propagated in type inference and the like are best handled by the language specification. Even this book has limits in terms of how deep it needs to go. Although you could use all of the preceding information and ignore what the CLR does with tuples, you'll be able to do more with tuples and better understand the behavior if you dig a bit deeper and find out how the compiler translates all of these rules into IL.

We've covered an awful lot of ground already. If you haven't tried writing code using tuples yet, now is a good time to do so. Take a break from the book and see if you can get a feel for tuples before learning how they're implemented.

11.4 Tuples in the CLR

Although in theory the C# language isn't tied to .NET, the reality is that every implementation I've seen at least attempts to look like the regular .NET Framework to some extent, even if it's compiled ahead of time and runs on a non-PC-desktop device. The C# language specification makes certain requirements of the final environment, including that certain types are available. At the time of this writing, there isn't a C# 7 specification, but I envision that when it's introduced, it'll require the types described in this section in order to use tuples.

Unlike anonymous types, in which each unique sequence of property names within an assembly causes the compiler to generate a new type, tuples don't require any extra types to be generated by the compiler. Instead, it uses a new set of types from the framework. Let's meet them now.

11.4.1 Introducing `System.ValueTuple<...>`

Tuples in C# 7 are implemented using the `System.ValueTuple` family of types. These types live in the `System.ValueTuple.dll` assembly, which is part of .NET Standard 2.0 but not part of any older .NET Framework releases. You can use it when targeting older frameworks by adding a dependency to the `System.ValueTuple` NuGet package.

There are nine `ValueTuple` structs with generic arities of 0 through 8:

- `System.ValueTuple` (nongeneric)
- `System.ValueTuple<T1>`
- `System.ValueTuple<T1, T2>`
- `System.ValueTuple<T1, T2, T3>`
- `System.ValueTuple<T1, T2, T3, T4>`
- `System.ValueTuple<T1, T2, T3, T4, T5>`
- `System.ValueTuple<T1, T2, T3, T4, T5, T6>`
- `System.ValueTuple<T1, T2, T3, T4, T5, T6, T7>`
- `System.ValueTuple<T1, T2, T3, T4, T5, T6, T7, TRest>`

For the moment, we're going to ignore the first two and the last one, although I talk about the latter in sections 11.4.7 and 11.4.8. That leaves us with the types with generic arities between 2 and 7 inclusive. (Realistically, those are the ones you're most likely to use anyway.)

A description of any particular `ValueTuple<...>` type is very much like the description of tuple types from earlier: it's a value type with public fields. The fields are called `Item1`, `Item2`, and so on, as far as `Item7`. The arity-8 tuple's final field is called `Rest`.

Anytime you use a C# tuple type, it's mapped onto a `ValueTuple<...>` type. That mapping is pretty obvious when the C# tuple type doesn't have any element names; `(int, string, byte)` is mapped to `ValueTuple<int, string, byte>`, for example. But what about the optional element names in C# tuple types? Generic types are generic only in their type parameters; you can't magically give two constructed types different field names. How does the compiler handle this?

11.4.2 Element name handling

Effectively, the C# compiler ignores the names for the purposes of mapping C# tuple types to CLR `ValueTuple<...>` types. Although `(int, int)` and `(int x, int y)` are distinct types from a C# language perspective, they both map onto `ValueTuple<int, int>`. The compiler then maps any uses of element names to the relevant `ItemN` name. Figure 11.5 shows the effective translation of C# with a tuple literal into C#, which refers only to the CLR types.

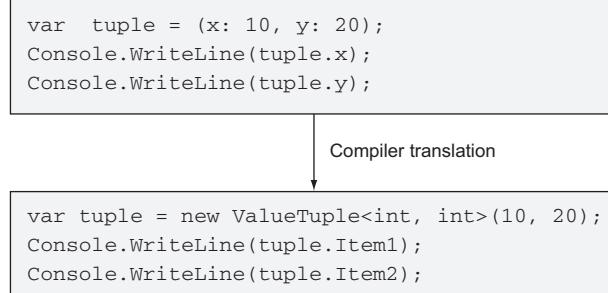


Figure 11.5 Compiler translation of tuple type handling into use of `ValueTuple`

Notice that the lower half of figure 11.5 has lost the names. For local variables like this, they're used only at compile time. The only trace of them at execution time would be in the PDB file created to give the debugger more information. What about element names that are visible outside the relatively small context of a method?

ELEMENT NAMES IN METADATA

Think back to the `MinMax` method you've used several times in this chapter. Suppose you make that method public as part of a whole package of aggregating methods to complement LINQ to Objects. It'd be a real shame to lose the readability afforded by the tuple element names, but you now know that the CLR return type of the method can't propagate them. Fortunately, the compiler can use the same technique that's already in place for other features that aren't directly supported by the CLR, such as `out` parameters and default parameter values; attributes to the rescue!

In this case, the compiler uses an attribute called `TupleElementNamesAttribute` (in the same namespace as many similar attributes: `System.Runtime.CompilerServices`) to encode the element names in the assembly. For example, a public `MinMax` method declaration could be represented in C# 6 as follows:

```
[return: TupleElementNames(new[] {"min", "max"})]
public static ValueTuple<int, int> MinMax(IEnumerable<int> numbers)
```

The C# 7 compiler won't let you compile that code. The compiler gives an error telling you to use tuple syntax directly. But compiling the same code with the C# 6 compiler results in an assembly you can use from C# 7, and the elements of the returned tuple will be available by name.

The attribute gets a bit more complicated when nested tuple types are involved, but it's unlikely that you'll ever need to interpret the attribute directly. It's just worth being aware that it exists and that's how the element names are communicated outside local variables. The attributes are emitted by the C# compiler even for private members, even though it could probably make do without them. I suspect it's considerably simpler to treat all members the same way regardless of their access modifiers.

NO ELEMENT NAMES AT EXECUTION TIME

In case it isn't obvious from everything that's gone before, a tuple value has no concept of element names at execution time. If you call `GetType()` on a tuple value, you'll get a `ValueTuple<...>` type with the appropriate element types, but any element names you have in your source code will be nowhere in sight. If you step through code and the debugger displays element names, that's because it's used extra information to work out the original element names; it's not something the CLR knows about directly.

NOTE This approach may feel familiar to Java developers. It's similar to the way Java handles generics with type information that isn't present at execution time. In Java, there's no such thing as an `ArrayList<Integer>` object or an `ArrayList<String>` object; they're just `ArrayList` objects. That's proved painful in Java, but the element names for tuples are less fundamentally

important than type arguments in generics, so hopefully it won't end up causing the same kind of problems.

Element names exist for tuples within the C# language, but not in the CLR. What about conversions?

11.4.3 Tuple conversion implementations

The types in the `ValueTuple` family don't provide *any* conversions as far as the CLR is concerned. They wouldn't be able to; the conversions that the C# language provides couldn't be expressed in the type information. Instead, the C# compiler creates a new value when it needs to, performing appropriate conversions on each element. Here are two examples of conversions, one implicit (using the implicit conversion from `int` to `long`) and one explicit (using the explicit conversion from `int` to `byte`):

```
(int, string) t1 = (300, "text");
(long, string) t2 = t1;
(byte, string) t3 = ((byte, string)) t1;
```

The compiler generates code as if you'd written this:

```
var t1 = new ValueTuple<int, string>(300, "text");
var t2 = new ValueTuple<long, string>(t1.Item1, t1.Item2);
var t3 = new ValueTuple<byte, string>((byte) t1.Item1, t1.Item2);
```

That example deals only with the conversions between tuple types that you've already seen, but the conversions for tuple literals to tuple types work in exactly the same way: any conversion required from an element expression to the target element type is just performed as part of calling the appropriate `ValueTuple<...>` constructor.

You've now learned about everything the compiler needs in order to provide tuple syntax, but the `ValueTuple<...>` types provide more functionality to make them easy to work with. Given how general they are, they can't do much, but the `ToString()` method has a readable output, and there are multiple options for comparing them. Let's see what's available.

11.4.4 String representations of tuples

The string representation of a tuple is like a tuple literal in C# source code: a sequence of values that is separated by commas and enclosed in parentheses. There's no fine-tuned control of this output; if you use a `(DateTime, DateTime)` tuple to represent a date interval, for example, you can't pass in a format string to indicate that you want the elements to be formatted just as dates. The `ToString()` method calls `ToString()` on each non-null element and uses an empty string for each null element.

As a reminder, none of the names you've provided to the tuple elements are known at execution time, so they can't appear in the results of calling `ToString()`. That can make it slightly less useful than the string representation of anonymous types, although if you're printing a lot of tuples of the same type, you'll be grateful for

the lack of repetition. One brief example is sufficient to demonstrate all of the preceding information:

```
var tuple = (x: (string) null, y: "text", z: 10);
Console.WriteLine(tuple.ToString());
```

The output of this snippet is as follows:

```
(, text, 10)
```

I've called `ToString()` explicitly here just to prove there's nothing else going on. You'd get the same output by calling `Console.WriteLine(tuple)`.

The string representation of tuples is certainly useful for diagnostic purposes, but it'd rarely be appropriate to display it directly in an end-user-facing application. You're likely to want to provide more context, specify format information for some types, and possibly handle null values more clearly.

11.4.5 Regular equality and ordering comparisons

Each `ValueTuple<...>` type implements `IEquatable<T>` and `IComparable<T>`, where `T` is the same as the type itself. For example, `ValueTuple<T1, T2>` implements `IEquatable<ValueTuple<T1, T2>>` and `IComparable<ValueTuple<T1, T2>>`.

Each type also implements the nongeneric `IComparable` interface and overrides the `object.Equals(object)` method in the natural way: `Equals(object)` will return `false` if it's passed an instance of a different type, and `CompareTo(object)` will throw an `ArgumentException` if it's passed an instance of a different type. Otherwise, each method delegates to its counterpart from `IEquatable<T>` or `IComparable<T>`.

Equality tests are performed element-wise using the default equality comparer for each element type. Similarly, element hash codes are computed using the default equality comparers, and then those hash codes are combined in an implementation-specific way to provide an overall hash code for the tuple. Ordering comparisons between tuples are performed element-wise too, with earlier elements being deemed more important in the comparisons than later ones, so `(1, 5)` compares as less than `(3, 2)`, for example.

These comparisons make tuples easy to work with in LINQ. Suppose you have a collection of `(int, int)` tuples representing `(x, y)` coordinates. You can use familiar LINQ operations to find distinct points in the list and order them. This is shown in the following listing.

Listing 11.9 Finding and ordering distinct points

```
var points = new[]
{
    (1, 2), (10, 3), (-1, 5), (2, 1),
    (10, 3), (2, 1), (1, 1)
};
```

```

var distinctPoints = points.Distinct();
Console.WriteLine($"{{distinctPoints.Count()}} distinct points");
Console.WriteLine("Points in order:");
foreach (var point in distinctPoints.OrderBy(p => p))
{
    Console.WriteLine(point);
}

```

The `Distinct()` call means that you see `(2, 1)` only once in the output. But the fact that equality is checked element-wise means that `(2, 1)` isn't equal to `(1, 2)`.

Because the first element in the tuple is considered the most important in ordering, our points will be sorted by their `x` coordinates; if multiple points have the same `x` coordinate, those will be sorted by their `y` coordinates. The output is therefore as follows:

```

5 distinct points
Points in order:
(-1, 5)
(1, 1)
(1, 2)
(2, 1)
(10, 3)

```

The regular comparisons provide no way of specifying how to compare each particular element. You can reasonably easily create your own custom implementations of `IEqualityComparer<T>` or `IComparer<T>` for specific tuple types, of course, but at that point you might want to consider whether it's worth implementing a fully custom type for the data you're trying to represent and avoid tuples entirely. Alternatively, in some cases it may be simpler to use structural comparisons.

11.4.6 Structural equality and ordering comparisons

In addition to the regular `IEquatable` and `IComparable` interfaces, each `ValueTuple` struct explicitly implements `IStructuralEquatable` and `IStructuralComparable`. These interfaces have existed since .NET 4.0 and are implemented by arrays and the `Tuple` family of immutable classes. I can't say I've ever used the interfaces myself, but that's not to claim they can't be used and used well. They mirror the regular APIs for equality and ordering, but each method takes a comparer that's intended to be used for the individual elements:

```

public interface IStructuralEquatable
{
    bool Equals(Object, IEqualityComparer);
    int GetHashCode(IEqualityComparer);
}

public interface IStructuralComparable
{
    int CompareTo(Object, IComparer);
}

```

The idea behind the interface is to allow composite objects to be compared for equality or ordering by performing pairwise comparisons with the given comparer. The regular generic comparisons implemented by `ValueTuple` types are statically type safe but relatively inflexible, as they always use default comparisons for the elements, whereas the structural comparisons are less type safe but provide extra flexibility. The following listing demonstrates this using strings and passing in a case-insensitive comparer.

Listing 11.10 Structural comparisons with a case-insensitive comparer

```

static void Main()
{
    var Ab = ("A", "b");
    var aB = ("a", "B");
    var aa = ("a", "a");
    var ba = ("b", "a");

    Compare(Ab, aB);
    Compare(aB, aa);
    Compare(aB, ba);
}

static void Compare<T>(T x, T y)
    where T : IStructuralEquatable, IStructuralComparable
{
    var comparison = x.CompareTo(
        y, StringComparer.OrdinalIgnoreCase);
    var equal = x.Equals(
        y, StringComparer.OrdinalIgnoreCase);

    Console.WriteLine(
        $"{x} and {y} - comparison: {comparison}; equal: {equal}");
}

```

Unconventional variable names that reflect the values

Performs a selection of interesting comparisons

Performs ordering and equality comparisons case insensitively

The output of listing 11.10 demonstrates that the comparisons are indeed performed pairwise in a case-insensitive way:

```
(A, b) and (a, B) - comparison: 0; equal: True
(a, B) and (a, a) - comparison: 1; equal: False
(a, B) and (b, a) - comparison: -1; equal: False
```

The benefit of this kind of comparison is that it's all down to composition: the comparer knows how to perform comparisons of only individual elements, and the tuple implementation delegates each comparison to the comparer. This is a little like LINQ, in which you express operations on individual elements but then ask them to be performed on collections.

This is all very well if you have tuples with elements that are all of the same type. If you want to perform structural comparisons on tuples with elements of different kinds, such as comparing `(string, int, double)` values, then you need to make sure your comparer can handle comparing strings, comparing integers, and comparing doubles. It'd only need to compare two values of the same type in each comparison, however.

The `ValueTuple` implementations still allow only tuples with the same type arguments to be compared; if you compare a `(string, int)` with an `(int, string)`, for example, an exception will be thrown immediately and before any elements are compared. An example of such a comparer is beyond the scope of this book, but the sample code contains a sketch (`CompoundEqualityComparer`) that should be a good starting point if you need to implement something similar for production code.

That concludes our coverage of the arity-2 to arity-7 `ValueTuple<...>` types, but I did mention that I'd come back to the other three types you saw in section 11.4.1. First, let's look at `ValueTuple<T1>` and `ValueTuple<T1, T2, T3, T4, T5, T6, T7, TRest>`, which are more closely related than you might expect.

11.4.7 Wombles and large tuples

A single-value tuple (`ValueTuple<T1>`), affectionately known as a *womble* by the C# team, can't be constructed on its own using tuple syntax, but it can be part of another tuple. As described earlier, generic `ValueTuple` structs exist with only up to eight type parameters. What should the C# compiler do if it's presented with a tuple literal with more than eight elements? It uses the `ValueTuple<...>` with arity 8 with the first seven type arguments corresponding to the first seven types from the tuple literal and the final element being a nested tuple type for the remaining elements. If you have a tuple literal with exactly eight `int` elements, the type involved will be as follows:

```
ValueTuple<int, int, int, int, int, int, int, ValueTuple<int>>
```

There's the womble, highlighted in bold. The `ValueTuple<...>` with arity 8 is especially designed for this usage; the final type argument (`TRest`) is constrained to be a value type, and, as I mentioned at the start of section 11.4.1, there's no `Item8` field. Instead, there's a `Rest` field.

It's important that the last element of an arity-8 `ValueTuple<...>` is always expected to be a tuple with more elements rather than a final individual element to avoid ambiguity. For example, a tuple type like this

```
ValueTuple<A, B, C, D, E, F, G, ValueTuple<H, I>>
```

could be treated as a C#-syntax type `(A, B, C, D, E, F, G, H, I)` with arity 9 or a type `(A, B, C, D, E, F, G, (H, I))` with arity 8 and the final element being a tuple type.

As a developer, you don't need to worry about all of this, because the C# compiler allows you to use `ItemX` names for *all* the elements in a tuple regardless of the number of elements and whether you've used the tuple syntax or explicitly referred to `ValueTuple`. For example, consider a rather long tuple:

```
var tuple = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16);
Console.WriteLine(tuple.Item16);
```

That's perfectly valid code, but the `tuple.Item16` expression is converted by the compiler into `tuple.Rest.Rest.Item2`. If you want to use the real field names, you can certainly do so; I just wouldn't advise it. Now from huge tuples to the exact opposite.

11.4.8 The nongeneric `ValueTuple` struct

If the womple sounded slightly silly to start with, the *nuple*—a nongeneric tuple, one without any elements at all—sounds even more pointless. You might have expected the nongeneric `ValueTuple` to be a static class, like the nongeneric `Nullable` class, but it's a struct and looks for all the world like the other tuple structs, other than not having any data. It implements all the interfaces described earlier in this section, but every nuple value is equal (in both plain equality and ordering senses) to every other nuple value, which makes sense, as there's nothing to differentiate them from each other.

It does have static methods that would be useful for creating `ValueTuple<...>` values if we didn't have tuple literals. Those methods will primarily be useful if you want to use tuple types from C# 6 or from another language that doesn't have built-in support, and you want to use type inference for the element types. (Remember, when you call a constructor, you always have to specify all the type arguments, which can be annoying.) For example, to construct an `(int, int)` value tuple in C# 6 using type inference, you could use this:

```
var tuple = ValueTuple.Create(5, 10);
```

The C# team has hinted that there may be future places where nuples will be useful with pattern matching and decomposition, but it's more of a placeholder than anything else at the moment.

11.4.9 Extension methods

The `System.TupleExtensions` static class is provided in the same assembly as the `System.ValueTuple` types. It contains extension methods on the `System.Tuple` and `System.ValueTuple` types. There are three kinds of methods:

- `Deconstruct`, which extends the `Tuple` types
- `ToValueTuple`, which extends the `Tuple` types
- `ToTuple`, which extends the `ValueTuple` types

Each kind of method is overloaded 21 times by generic arity using the same pattern you saw previously to handle arities of 8 or more. You'll look at `Deconstruct` in chapter 12, but `ToValueTuple` and `ToTuple` do exactly what you'd expect them to: they convert between the .NET 4.0 era immutable reference type tuples and the new mutable value type tuples. I expect these to be useful primarily when working with legacy code using `Tuple`.

Phew! That's just about everything I think is worth knowing about the types involved in implementing tuples on the CLR. Next, we're going to consider your other options: if you're thinking about using a tuple, you should be aware that's only one of the tools in your box, and it isn't always the most appropriate one to reach for.

11.5 Alternatives to tuples

It may seem trite to remind you of this, but every option you've ever used in the past for a bag of variables is still valid. You don't *have* to use the C# 7 tuples anywhere. This section briefly looks at the pros and cons of the other options.

11.5.1 System.Tuple<...>

The .NET 4 `System.Tuple<...>` types are immutable reference types, although the element types within them may be mutable. You can think of it as being immutable in a shallow fashion, just like `readonly` fields.

The biggest downside here is the lack of any kind of language integration. Old-school tuples are harder to create, the types are more long-winded to specify, the conversions I describe in section 11.3 simply aren't present, and most important, you can use only the `ItemX` naming style. Even though the names attached to C# 7 tuples are compile-time only, they still make a huge difference in usability.

Beyond this, reference type tuples feel like fully fledged objects instead of bags of values, which can be good or bad depending on the context. They're usually less convenient to work with, but it's certainly more efficient to copy a single reference to a large `Tuple<...>` object than it is to copy a `ValueTuple<...>`, which involves copying all the element values. This also has implications on safe multithreading: copying a reference is atomic, whereas copying a value tuple isn't.

11.5.2 Anonymous types

Anonymous types were introduced as part of LINQ, and that remains their primary use case in my experience. You could use them for regular variables within a method, but I can't remember ever seeing that usage in production code.

Most of the nice features of anonymous types are also present in C# 7 tuples: named elements, natural equality, and a clear string representation. The main problem with anonymous types is precisely that they're anonymous; you can't return them from methods or properties without losing all the type safety. (You basically have to use `object` or `dynamic`. The information is still there at execution time, but the compiler doesn't know about it.) C# 7 tuples don't have that problem. It's fine to return a tuple from a method, as you've seen.

I can see four advantages of anonymous types over tuples:

- In C# 7.0, projection initializers that provide both a name and a value in a single identifier are simpler than tuples; compare `new { p.Name, p.Age }` and `(name: p.Name, age: p.Age)`, for example. This is addressed in C# 7.1, as the tuple element names can be inferred, leading to compact representations such as `(p.Name, p.Age)`.
- The use of names within the string representation of anonymous types can be handy for diagnostic purposes.
- Anonymous types are supported by out-of-process LINQ providers (to databases and so on). Tuple literals can't currently be used within expression trees, making the value proposition significantly weaker.
- Anonymous types can be more efficient in some contexts due to a single reference being passed through a pipeline. In most cases, I wouldn't expect this to be a problem at all, and the fact that tuples don't create any objects for the garbage collector to clean up is an efficiency benefit in the other direction, of course.

Within LINQ to Objects, I expect to use tuples extensively, particularly when using C# 7.1 and its inferred tuple element names.

11.5.3 Named types

Tuples are just bags of variables. There's no encapsulation; no meaning is attached to them other than what you decide to do with them. Sometimes that's exactly what you want, but be careful of taking this too far. Consider a `(double, double)`. That could be used as

- 2D Cartesian coordinates (`x, y`)
- 2D polar coordinates (`radius, angle`)
- 1D start/end pair
- Any number of other things

Each of these use cases would have different operations on it when modeled as a first-class type. You wouldn't need to worry about names not being propagated or accidentally using Cartesian coordinates as polar coordinates, for example.

If you need the grouping of values only temporarily, or if you're prototyping and you're not sure what you'll need, tuples are great. But if you find you're using the same tuple shape in several places in your code, I'd recommend replacing it with a named type.

NOTE A Roslyn code analyzer to automate most of this, using tuple element names to detect different usages, could be wonderful. I don't know of any such tool at the moment, unfortunately.

With that background of alternative options, let's round off this chapter with some more detailed recommendations about where tuples might be useful.

11.6 Uses and recommendations

First, it's important to remember that language support for tuples is new within C# 7. Any suggestions here are the result of *thinking* about tuples rather than extensive *use* of tuples. Reason can get you so far, but it doesn't give much insight into the actual experience. My expectations about when I'd use new language features have proved somewhat incorrect in the past, so take everything here with a grain of salt. That said, hopefully it at least provides some food for thought.

11.6.1 Nonpublic APIs and easily changed code

Until the community in general has more experience with tuples and best practices have been established through hard-won battle scars, I'd avoid using tuples in public APIs, including protected members for types that can be derived from in other assemblies. If you're in the lucky situation where you control (and can modify arbitrarily) all the code that interacts with yours, you can be more speculative. But you don't want to put yourself in a situation where you return a tuple from a public method just because

it's easy to do, only to discover later that you really wanted to encapsulate those values more thoroughly. A named type takes more design and implementation work, but the result is unlikely to be harder for the caller to use. Tuples are mostly convenient for the implementer rather than the caller.

My current preference is to go even further and use tuples only as an implementation detail within a type. I'm comfortable returning a tuple from a private method, but I'd shy away from doing so from an internal method in production code. In general, the more localized the decision, the easier it is to change your mind and the less you have to think about it.

11.6.2 Local variables

Tuples are primarily designed to allow multiple values to be returned from a method without using `out` parameters or a dedicated return type. That doesn't mean that's the only place you can use them, though.

It's not unusual within a method to have natural groups of variables. You can often tell this when you look at the variables if they have a common prefix. For example, listing 11.11 shows a method that might occur in a game to display the highest-scoring player for a particular date. Although LINQ to Objects has a `Max` method that'll return the highest value for a projection, there's nothing that will return the original sequence element associated with that value.

NOTE An alternative is to use `OrderByDescending(...).FirstOrDefault()`, but that's introducing sorting when you need to find only a single value. The `MoreLinq` package has the `MaxBy` method, which addresses this deficiency. Another alternative to keeping two variables is to keep a single `highestGame` variable and use the `Score` property of that in the comparison. In more complex cases, that may not be as feasible.

Listing 11.11 Displaying the highest-scoring player for a date

```
public void DisplayHighScoreForDate(LocalDate date)
{
    var filteredGames = allGames.Where(game => game.Date == date);
    string highestPlayer = null;
    int highestScore = -1;
    foreach (var game in filteredGames)
    {
        if (game.Score > highestScore)
        {
            highestPlayer = game.PlayerName;
            highestScore = game.Score;
        }
    }
    Console.WriteLine(highestPlayer == null
        ? "No games played"
        : $"Highest score was {highestScore} by {highestPlayer}");
}
```

Here you have four local variables, including the parameter:

- date
- filteredGames
- highestPlayer
- highestScore

The last two of these are tightly related to each other; they're initialized at the same time and changed together. This suggests you could *consider* using a tuple variable instead, as in the following listing.

Listing 11.12 A refactoring to use a tuple local variable

```
public void DisplayHighScoreForDate(LocalDate date)
{
    var filteredGames = allGames.Where(game => game.Date == date);
    (string player, int score) highest = (null, -1);
    foreach (var game in filteredGames)
    {
        if (game.Score > highest.score)
        {
            highest = (game.PlayerName, game.Score);
        }
    }
    Console.WriteLine(highest.player == null
        ? "No games played"
        : $"Highest score was {highest.score} by {highest.player}");
}
```

The changes are shown in bold. Is this better? Maybe. Philosophically, it's exactly the same code, when you think about a tuple as just a collection of variables. It feels slightly cleaner to me, because it reduces the number of concepts the method is considering at the top level. Obviously, in the kind of simplistic examples that are applicable to books, differences in clarity are likely to be small. But if you have a complicated method that's resistant to refactoring into multiple smaller methods, tuple local variables could make a more significant difference. The same kind of consideration makes sense for fields, too.

11.6.3 Fields

Just as local variables sometimes cluster together naturally, so do fields. Here's an example from Noda Time in `PrecalculatedDateTimeZone`:

```
private readonly ZoneInterval[] periods;
private readonly IZoneIntervalMapWithMinMax tailZone;
private readonly Instant tailZoneStart;
private readonly ZoneInterval firstTailZoneInterval;
```

I'm not going to explain the meaning of all these fields, but hopefully it's reasonably obvious that the last three of them relate to a tail zone. We could consider changing this to use two fields instead, one of which is a tuple:

```
private readonly ZoneInterval[] periods;
private readonly
    (IZoneIntervalMapWithMinMax intervalMap,
    Instant start,
    ZoneInterval firstInterval) tailZone;
```

The rest of the code can then refer to `tailZone.start`, `tailZone.intervalMap`, and so forth. Note that because the `tailZone` variable is declared to be `readonly`, assignments to the individual elements are invalid except in the constructor. A few limitations and caveats exist:

- The elements of the tuple can still be assigned individually in the constructor, but there's no warning if you initialize some elements but not all of them. For example, if you forgot to initialize `tailZoneStart` in the original code, you'd see a warning, but there's no equivalent warning if you forget to initialize `tailZone.start`.
- Either the whole tuple field is read-only or none of it is. If you have a group of related fields, some of which are read-only and some of which aren't, you either have to forego the read-only aspect or not use this technique. At that point, I'd usually just not use tuples.
- If some of the fields are automatically generated fields backing automatically implemented properties, you'd have to write full properties to use the tuple. Again, at that point I'd skip the tuple.

Finally, one aspect of tuples that may not be obvious is their interaction with dynamic typing.

11.6.4 Tuples and dynamic don't play together nicely

I don't use `dynamic` much myself anyway, and I suspect that good uses of dynamic typing and good uses of tuples won't have much of an intersection. It's worth being aware of two issues around element access, however.

THE DYNAMIC BINDER DOESN'T KNOW ABOUT ELEMENT NAMES

Remember that element names are mostly a compile-time concern. Mix that with the way dynamic binding happens only at execution times, and I suspect you can see what's coming. As a simple example, consider the following code:

```
dynamic tuple = (x: 10, y: 20);
Console.WriteLine(tuple.x);
```

At first glance, it sounds reasonable to expect this to print 10, but an exception is thrown:

```
Unhandled Exception: Microsoft.CSharp.RuntimeBinder.RuntimeBinderException:
'System.ValueTuple<int,int>' does not contain a definition for 'x'
```

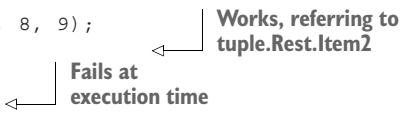
Although this is unfortunate, it'd require significant gymnastics for element name information to be preserved for the dynamic binder to make this work. I'm not expecting this to change. If you modify the code to print `tuple.Item1` instead, that's fine. At least, it's fine for the first seven elements.

THE DYNAMIC BINDER DOESN'T (CURRENTLY) KNOW ABOUT HIGH ELEMENT NUMBERS

In section 11.5.4, you saw how the compiler handles tuples with more than seven elements. The compiler uses the arity-8 `ValueTuple<...>` with a final element that contains another tuple accessed via the `Rest` field instead of an `Item8` field. In addition to transforming the type itself, the compiler transforms numbered element access; source code referring to `tuple.Item9` refers to `tuple.Rest.Item2` in the generated IL, for example.

At the time of this writing, the dynamic binder isn't aware of this, so again you'll see an exception where the same code would be fine with compile-time binding. As an example, you can easily test and play with this yourself:

```
var tuple = (1, 2, 3, 4, 5, 6, 7, 8, 9);
Console.WriteLine(tuple.Item9);
dynamic d = tuple;
Console.WriteLine(d.Item9);
```



Unlike the previous issue, this could be fixed by making the dynamic binder smarter. But the execution-time behavior will then depend on which version of the dynamic binder your application ends up using. Usually, a reasonably clean separation exists between which version of the compiler you use and which assembly and framework versions you use. Requiring a particular version of the dynamic binder would certainly muddy the waters somewhat.

Summary

- Tuples act as bags of elements with no encapsulation.
- Tuples in C# 7 have distinct language and CLR representations.
- Tuples are value types with public, mutable fields.
- C# tuples support flexible element names.
- The CLR `ValueTuple<...>` structs always use element names of `Item1`, `Item2`, and so forth.
- C# provides conversions for tuple types and tuple literals.

12

Deconstruction and pattern matching

This chapter covers

- Deconstructing tuples into multiple variables
- Deconstructing nontuple types
- Applying pattern matching in C# 7
- Using the three kinds of patterns introduced in C# 7

In chapter 11, you learned that tuples allow you to compose data simply without having to create new types and allowing one variable to act as a bag of other variables. When you used the tuples—for example, to print out the minimum value from a sequence of integers and then print out the maximum—you extracted the values from the tuple one at a time.

That certainly works, and in many cases it's all you need. But in plenty of cases, you'll want to break a composite value into separate variables. This operation is called *deconstruction*. That composite value may be a tuple, or it could be of another

type—`KeyValuePair`, for example. C# 7 provides simple syntax to allow multiple variables to be declared or initialized in a single statement.

Deconstruction occurs in an unconditional way just like a sequence of assignments. Pattern matching is similar, but in a more dynamic context; the input value has to match the pattern in order to execute the code that follows it. C# 7 introduces pattern matching in a couple of contexts and a few kinds of patterns, and there will likely be more in future releases. We'll start building on chapter 11 by deconstructing the tuples you've just created.

12.1 Deconstruction of tuples

C# 7 provides two flavors of deconstruction: one for tuples and one for everything else. They follow the same syntax and have the same general features, but talking about them in the abstract can be confusing. We'll look at tuples first, and I'll call out anything that's tuple specific. In section 12.2, you'll see how the same ideas are applied to other types. Just to give you an idea of what's coming, the following listing shows several features of deconstruction, each of which you'll examine in more detail.

Listing 12.1 Overview of deconstruction using tuples

```
var tuple = (10, "text");           ← Creates a tuple of type (int, string)
var (a, b) = tuple;               ← Deconstructs to new variables a, b implicitly
(int c, string d) = tuple;       ← Deconstructs to new variables c, d explicitly
int e;
string f;                         ← Deconstructs to existing variables
(e, f) = tuple;                  ← Proves that deconstruction works
Console.WriteLine($"a: {a}; b: {b}");
Console.WriteLine($"c: {c}; d: {d}");
Console.WriteLine($"e: {e}; f: {f}");
```

I suspect that if you were shown that code and told that it would compile, you'd already be able to guess the output, even if you hadn't read anything about tuples or deconstruction before:

```
a: 10; b: text
c: 10; d: text
e: 10; f: text
```

All you've done is declared and initialized the six variables `a`, `b`, `c`, `d`, `e`, and `f` in a new way that takes less code than it would've before. This isn't to diminish the usefulness of the feature, but this time there's relatively little subtlety to go into. In all cases, the operation is as simple as copying a value out of the tuple into a variable. It doesn't associate the variable with the tuple; changing the variable later won't change the tuple, or vice versa.

Tuple declaration and deconstruction syntax

The language specification regards deconstruction as closely related to other tuple features. Deconstruction syntax is described in terms of a *tuple expression* even when you're not deconstructing tuples (which you'll see in section 12.2). You probably don't need to worry too much about that, but you should be aware of potential causes for confusion. Consider these two statements:

```
(int c, string d) = tuple;
(int c, string d) x = tuple;
```

The first uses deconstruction to declare two variables (`c` and `d`); the second is a declaration of a single variable (`x`) of tuple type (`int c, string d`). I don't think this similarity was a design mistake, but it can take a little getting used to just like expression-bodied members looking like lambda expressions.

Let's start by looking in more detail at the first two parts of the example, where you declare and initialize in one statement.

12.1.1 Deconstruction to new variables

It's always been feasible to declare multiple variables in a single statement, but only if they were of the same type. I've typically stuck to a single declaration per statement for the sake of readability. But when you can declare and initialize multiple variables in a single statement, and the initial values all have the same source, that's neat. In particular, if that source is the result of a function call, you can avoid declaring an extra variable just to avoid making multiple calls.

The syntax that's probably simplest to understand is the one in which each variable is explicitly typed—the same syntax as for a parameter list or tuple type. To clarify my preceding point about the extra variable, the following listing shows a tuple as a result of a method call being deconstructed into three new variables.

Listing 12.2 Calling a method and deconstructing the result into three variables

```
static (int x, int y, string text) MethodReturningTuple() => (1, 2, "t");

static void Main()
{
    (int a, int b, string name) = MethodReturningTuple();
    Console.WriteLine($"a: {a}; b: {b}; name: {name}");
}
```

The benefit isn't as obvious until you consider the equivalent code without using deconstruction. This is what the compiler is transforming the preceding code into:

```
static void Main()
{
    var tmp = MethodReturningTuple();
    int a = tmp.x;
```

```

int b = tmp.y;
string name = tmp.text;

Console.WriteLine($"a: {a}; b: {b}; name: {name}");
}

```

The three declaration statements don't bother me too much, although I do appreciate the brevity of the original code, but the `tmp` variable really niggles. As its name suggests, it's there only temporarily; its sole purpose is to remember the result of the method call so it can be used to initialize the three variables you really want: `a`, `b`, and `name`. Even though you want `tmp` only for that bit of code, it has the same scope as the other variables, which feels messy to me. If you want to use implicit typing for some variables but explicit typing for others, that's fine too, as shown in figure 12.1.

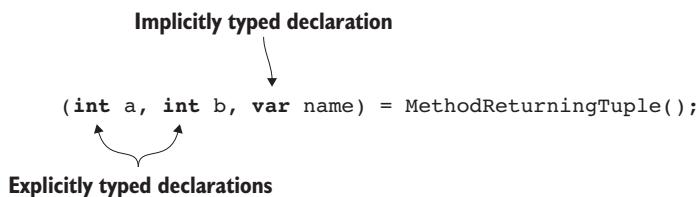


Figure 12.1 Mixing implicit and explicit typing in deconstruction

This is particularly useful if you want to specify a different type than the element type in the original tuple using an implicit conversion for elements where required; see figure 12.2.

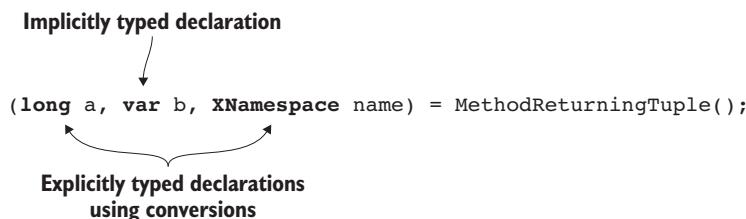


Figure 12.2 Deconstruction involving implicit conversions

If you're happy to use implicit typing for all the variables, C# 7 has shorthand to make it simple; just use `var` before the list of names:

```
var (a, b, name) = MethodReturningTuple();
```

This is equivalent to using `var` for each variable inside the parameter list, and that in turn is equivalent to explicitly specifying the inferred type based on the type of the value being assigned. Just as with regular implicitly typed variable declarations, using `var` doesn't make your code dynamically typed; it just makes the compiler infer the type.

Although you can mix and match between implicit typing and explicit typing in terms of the types specified within the brackets, you can't use `var` before the variable list and then provide types for some variables:

```
var (a, long b, name) = MethodReturningTuple();
```

Invalid: mixture of “inside and outside” declarations

A SPECIAL IDENTIFIER: `_` DISCARDS

C# 7 has three features that allow new places to introduce local variables:

- Deconstruction (this section and 12.2)
- Patterns (sections 12.3 to 12.7)
- Out variables (section 14.2)

In all these cases, specifying a variable name of `_` (a single underscore) has a special meaning. It's a *discard*, which means “I don't care about the result. I don't even want it as a variable at all—just get rid of it.” When a discard is used, it doesn't introduce a new variable into scope. You can use multiple discards instead of specifying different variable names for multiple variables you don't care about.

Here's an example of discards in tuple deconstruction:

```
var tuple = (1, 2, 3, 4);
var (x, y, _, _) = tuple;
Console.WriteLine(_);
```

Tuple with
four elements

Deconstructs the tuple but keeps
only the first two elements

Error CS0103: The name '_ doesn't
exist in the current context'

If you already have a variable called `_` in scope (declared with a regular variable declaration), you can still use discards in deconstruction to an otherwise new set of variables, and the existing variable will remain untouched.

As you saw in our original overview, you don't have to declare new variables to use deconstruction. Deconstruction can act as a sequence of assignments instead.

12.1.2 Deconstruction assignments to existing variables and properties

The previous section explained most of our original overview example. In this section, we'll look at this part of the code instead:

```
var tuple = (10, "text");
int e;
string f;
(e, f) = tuple;
```

In this case, the compiler isn't treating the deconstruction as a sequence of declarations with corresponding initialization expressions; instead, it's just a sequence of

assignments. This has the same benefit in terms of avoiding temporary variables that you saw in the previous section. The following listing gives an example using the same `MethodReturningTuple()` that you used before.

Listing 12.3 Assignments to existing variables using deconstruction

```
static (int x, int y, string text) MethodReturningTuple() => (1, 2, "t");

static void Main()
{
    int a = 20;
    int b = 30;
    string name = "before";
    Console.WriteLine($"a: {a}; b: {b}; name: {name}");
    (a, b, name) = MethodReturningTuple();           ← Declares, initializes, and
                                                    uses three variables

    Console.WriteLine($"a: {a}; b: {b}; name: {name}"); ← Assigns to all three variables
                                                    using deconstruction
}
                                                    ← Displays the
                                                    new values
```

So far, so good, but the feature doesn't stop with the ability to assign to local variables. Any assignment that would be valid as a separate statement is also valid using deconstruction. That can be an assignment to a field, a property, or an indexer, including working on arrays and other objects.

Declarations or assignments: Not a mixture

Deconstruction allows you to either declare and initialize variables or execute a sequence of assignments. You can't mix the two. For example, this is invalid:

```
int x;
(x, int y) = (1, 2);
```

It's fine for the assignments to use a variety of targets, however: some existing local variables, some fields, some properties, and so on.

In addition to regular assignments, you can assign to a discard (the `_` identifier), thereby effectively throwing away the value if there's nothing called `_` in scope. If you do have a variable named `_` in scope, deconstruction assigns to it as normal.

Using `_` in deconstruction: Assign or discard?

This looks a little confusing at first: sometimes deconstruction to `_` when there's an existing variable with that name changes the value, and sometimes it discards it. You can avoid this confusion in two ways. The first is to look at the rest of the deconstruction to see whether it's introducing new variables (in which case `_` is a discard) or assigning values to existing variables (in which case `_` is assigned a new value like the other variables).

The second way to avoid confusion is to not use `_` as a local variable name.

In practice, I expect almost all uses of assignment deconstruction to target either local variables or fields and properties of `this`. In fact, there's a neat little technique you can use in constructors that makes the expression-bodied constructors introduced in C# 7 even more useful. Many constructors assign values to properties or fields based on the constructor parameters. You can perform all those assignments in a single expression if you collect the parameters into a tuple literal first, as shown in the next listing.

Listing 12.4 Simple constructor assignments using deconstruction and a tuple literal

```
public sealed class Point
{
    public double X { get; }
    public double Y { get; }

    public Point(double x, double y) => (X, Y) = (x, y);
}
```

I really like the brevity of this. I love the clarity of the mapping from constructor parameter to property. The C# compiler even recognizes it as a pattern and avoids constructing a `ValueTuple<double, double>`. Unfortunately, it still requires a dependency on `System.ValueTuple.dll` to build, which is enough to put me off using it unless I'm also using tuples somewhere else in the project or targeting a framework that already includes `System.ValueTuple`.

Is this idiomatic C#?

As I've described, this trick has pros and cons. It's a pure implementation detail of the constructor; it doesn't even affect the rest of the class body. If you decide to embrace this style and then decide you don't like it, removing it should be trivial. It's too early to say whether this will catch on, but I hope so. I'd be wary as soon as the tuple literal needs to be more than just the exact parameter values, though. Even adding a single precondition tips the balance in favor of a regular sequence of assignments, in my subjective opinion.

Assignment deconstruction has an extra wrinkle compared with declaration deconstruction in terms of ordering. Deconstruction that uses assignment has three distinct stages:

- 1 Evaluating the targets of the assignments
- 2 Evaluating the right-hand side of the assignment operator
- 3 Performing the assignments

Those three stages are performed in exactly that order. Within each stage, evaluation occurs in left-to-right source order, as normal. It's rare that this can make a difference, but it's possible.

TIP If you have to worry about this section in order to understand code in front of you, that's a strong code smell. When you *do* understand it, I urge you

to refactor it. Deconstruction has all the same caveats of using side effects within an expression but amplified because you have multiple evaluations to perform in each stage.

I'm not going to linger on this topic for long; a single example is enough to show the kind of problem you might see. This is by no means the worst example you might find, however. There are all kinds of things you could do in order to make this more convoluted. The following listing deconstructs a `(StringBuilder, int)` tuple into an existing `StringBuilder` variable and the `Length` property associated with that variable.

Listing 12.5 Deconstruction in which evaluation order matters

```
StringBuilder builder = new StringBuilder("12345");
StringBuilder original = builder;
(builder, builder.Length) =
    (new StringBuilder("67890"), 3);           | Keeps a reference to original
                                            | builder for diagnostic reasons
Console.WriteLine(original);          | Displays the contents of
Console.WriteLine(builder);          | the old and new builders
```

The middle line is the tricky one here. The key question to consider is which `StringBuilder` has its `Length` property set: the one that `builder` refers to originally or the new value assigned in the first part of the deconstruction? As I described earlier, all the targets for the assignments are evaluated first, before any assignments are performed. The following listing demonstrates this in a sort of exploded version of the same code in which the deconstruction is performed manually.

Listing 12.6 Slow-motion deconstruction to show evaluation order

```
StringBuilder builder = new StringBuilder("12345");
StringBuilder original = builder;
StringBuilder targetForLength = builder;           | Evaluates
                                                    | assignment targets
(StringBuilder, int) tuple =
    (new StringBuilder("67890"), 3);           | Evaluates the
                                                | tuple literal
builder = tuple.Item1;
targetForLength.Length = tuple.Item2;           | Performs the assignments
                                                | on the targets
Console.WriteLine(original);
Console.WriteLine(builder);
```

No extra evaluation is required when the target is just a local variable; you can assign directly to it. But assigning to a property of a variable requires evaluating that variable value as part of the first phase; that's why you have the `targetForLength` variable.

After the tuple has been constructed from the literal, you can assign the different items to your targets, making sure you use `targetForLength` rather than `builder`

when assigning the `Length` property. The `Length` property is set on the original `StringBuilder` with content 12345 rather than the new one with content 67890. That means the output of listings 12.5 and 12.6 is as follows:

```
123
67890
```

With that out of the way, there's one final—and rather more pleasant—wrinkle of tuple construction to talk about before moving on to nontuple deconstruction.

12.1.3 Details of tuple literal deconstruction

As I described in section 11.3.1, not all tuple literals have a type. For example, the tuple literal `(null, x => x * 2)` doesn't have a type because neither of its element expressions has a type. But you know it can be converted to type `(string, Func<int, int>)` because each expression has a conversion to the corresponding type.

The good news is that tuple deconstruction has exactly the same sort of “per element assignment compatibility” as well. This works for both declaration deconstructions and assignment deconstructions. Here's a brief example:

```
(string text, Func<int, int> func) =
    (null, x => x * 2);
(text, func) = ("text", x => x * 3);
```

This also works with deconstruction that requires an implicit conversion from an expression to the target type. For example, using our favorite “int constant within range of byte” example, the following is valid:

```
(byte x, byte y) = (5, 10);
```

Like many good language features, this is probably something you might have implicitly expected, but the language needs to be carefully designed and specified to allow it. Now that you've looked at tuple deconstruction fairly extensively, deconstruction of nontuples is relatively straightforward.

12.2 Deconstruction of nontuple types

Deconstruction for nontuple types uses a pattern-based¹ approach in the same way `async/await` does and `foreach` can. Just as any type with a suitable `GetAwaiter` method or extension method can be awaited, any type with a suitable `Deconstruct` method or extension method can be deconstructed using the same syntax as tuples. Let's start with deconstruction using regular instance methods.

¹ This is entirely distinct from the patterns coming up in section 12.3. Apologies for the terminology collision.

12.2.1 Instance deconstruction methods

It's simplest to demonstrate deconstruction with the `Point` class used in several examples now. You can add a `Deconstruct` method to it like this:

```
public void Deconstruct(out double x, out double y)
{
    x = X;
    y = Y;
}
```

Then you can deconstruct any `Point` to two double variables as in the following listing.

Listing 12.7 Deconstructing a Point to two variables

```
var point = new Point(1.5, 20);
var (x, y) = point;
Console.WriteLine($"x = {x}");
Console.WriteLine($"y = {y}");
```

Constructs an instance of point

Displays the two variable values

Deconstructs it to two variables of type double

The `Deconstruct` method's job is to populate the `out` parameters with the result of the deconstruction. In this case, you're just deconstructing to two double values. It's like a constructor in reverse, as the name suggests.

But wait; you used a neat trick with tuples to assign parameter values to properties in the constructor in a single statement. Can you do that here? Yes, you can, and personally, I love it. Here are both the constructor and the `Deconstruct` method so you can see the similarities:

```
public Point(double x, double y) => (X, Y) = (x, y);
public void Deconstruct(out double x, out double y) => (x, y) = (X, Y);
```

The simplicity of this is beautiful, at least after you've gotten used to it.

The rules of `Deconstruct` instance methods used for deconstruction are pretty simple:

- The method must be accessible to the code doing the deconstruction. (For example, if everything is in the same assembly, it's fine for `Deconstruct` to be an internal method.)
- It must be a `void` method.
- There must be at least two parameters. (You can't deconstruct to a single value.)
- It must be nongeneric.

You may be wondering why the design uses `out` parameters instead of requiring that `Deconstruct` is parameterless but has a tuple return type. The answer is that it's useful to be able to deconstruct to multiple sets of values, which is feasible with multiple methods, but you can't overload methods just on return type. To make this clearer, I'll use an example deconstructing `DateTime`, but of course, you can't add your own instance methods to `DateTime`. It's time to introduce extension deconstruction methods.

12.2.2 Extension deconstruction methods and overloading

As I briefly stated in the introduction, the compiler finds any Deconstruct methods that follow the relevant pattern, including extension methods. You can probably imagine what an extension method for deconstruction looks like, but the following listing gives a concrete example, using `DateTime`.

Listing 12.8 Using an extension method to deconstruct `DateTime`

```
static void Deconstruct(
    this DateTime dateTime,
    out int year, out int month, out int day) =>
    (year, month, day) =
        (dateTime.Year, dateTime.Month, dateTime.Day);

static void Main()
{
    DateTime now = DateTime.UtcNow;
    var (year, month, day) = now;           ← Deconstructs the current
    Console.WriteLine(                     date to year/month/day
        $"{{year:0000}}-{{month:00}}-{{day:00}}");   ← Displays the date using
}                                         the three variables
```

Extension method to deconstruct `DateTime`

As it happens, this is a private extension method declared in the same (static) class that you're using it from, but it'd more commonly be public or internal, just like most extension methods are.

What if you want to deconstruct a `DateTime` to more than just a date? This is where overloading is useful. You can have two methods with different parameter lists, and the compiler will work out which to use based on the number of parameters. Let's add another extension method to deconstruct a `DateTime` in terms of time as well as date and then use both our methods to deconstruct different values.

Listing 12.9 Using `Deconstruct` overloads

```
static void Deconstruct(
    this DateTime dateTime,
    out int year, out int month, out int day) =>
    (year, month, day) =
        (dateTime.Year, dateTime.Month, dateTime.Day);

static void Deconstruct(
    this DateTime dateTime,
    out int year, out int month, out int day,
    out int hour, out int minute, out int second) =>
    (year, month, day, hour, minute, second) =
        (dateTime.Year, dateTime.Month, dateTime.Day,
        dateTime.Hour, dateTime.Minute, dateTime.Second);

static void Main()
{
```

Deconstructs a date to year/month/day

Deconstructs a date to year/month/day/hour/minute/second

```

DateTime birthday = new DateTime(1976, 6, 19);
DateTime now = DateTime.UtcNow;

var (year, month, day, hour, minute, second) = now;           ← Uses the six-value
(year, month, day) = birthday;                                deconstructor
}
}                                                               ← Uses the three-value
deconstructor

```

You can use extension `Deconstruct` methods for types that already have instance `Deconstruct` methods, and they'll be used if the instance methods aren't applicable when deconstructing, just as for normal method calls.

The restrictions for an extension `Deconstruct` method follow naturally from those of an instance method:

- It has to be accessible to the calling code.
- Other than the first parameter (the target of the extension method), all parameters must be `out` parameters.
- There must be at least two such `out` parameters.
- The method may be generic, but only the receiver of the call (the first parameter) can participate in type inference.

The rules indicating when a method can and can't be generic deserve closer scrutiny, particularly because they also shed light on why you need to use a different number of parameters when overloading `Deconstruct`. The key lies in how the compiler treats the `Deconstruct` method.

12.2.3 Compiler handling of `Deconstruct` calls

When everything's working as expected, you can get away without thinking too much about how the compiler decides which `Deconstruct` method to use. If you run into problems, however, it can be useful to try to put yourself in the place of the compiler.

The timing you've already seen for tuple decomposition still applies when deconstructing with methods, so I'll focus on the method call itself. Let's take a somewhat concrete example, working out what the compiler does when faced with a deconstruction like this:

```
(int x, string y) = target;
```

I say this is a *somewhat* concrete example because I haven't shown what the type of `target` is. That's deliberate, because all you need to know is that it isn't a tuple type. The compiler expands this into something like this:

```
target.Deconstruct(out var tmpX, out var tmpY);
int x = tmpX;
string y = tmpY;
```

It then uses all the normal rules of method invocation to try to find the right method to call. I realize that the use of `out var` is something you haven't seen before. You'll look at it more closely in section 14.2, but all you need to know for now is that

it's declaring an implicitly typed variable using the type of the `out` parameter to infer the type.

The important thing to notice is that the types of the variables you've declared in the original code aren't used as part of the `Deconstruct` call. That means they can't participate in type inference. This explains three things:

- Instance `Deconstruct` methods can't be generic, because there's no information for type inference to use.
- Extension `Deconstruct` methods can be generic, because the compiler may be able to infer type arguments using `target`, but that's the only parameter that's going to be useful in terms of type inference.
- When overloading `Deconstruct` methods, it's the number of `out` parameters that's important, not their type. If you introduce multiple `Deconstruct` methods with the same number of `out` parameters, that's just going to stop the compiler from using any of them, because the calling code won't be able to tell which one you mean.

I'll leave it at that, because I don't want to make more of this than needed. If you run into problems that you can't understand, try performing the transformation shown previously, and it may well make things clearer.

That's everything you need to know about deconstruction. The rest of the chapter focuses on pattern matching, a feature that's theoretically entirely separate from deconstruction but has a similar feeling to it in terms of the tools available for using existing data in new ways.

12.3 *Introduction to pattern matching*

Like many other features, pattern matching is new to C# but not new to programming languages in general. In particular, functional languages often make heavy use of patterns. The patterns in C# 7.0 satisfy many of the same use cases but in a manner that fits in with the rest of the syntax of the language.

The basic idea of a pattern is to test a certain aspect of a value and use the result of that test to perform another action. Yes, that sounds just like an `if` statement, but patterns are typically used either to give more context for the condition or to provide more context within the action itself based on the pattern. Yet again, this feature doesn't allow you to do anything you couldn't do before; it just lets you express the same intention more clearly.

I don't want to go too far without giving an example. Don't worry if it seems a little odd right now; the aim is to give you a flavor. Suppose you have an abstract class `Shape` that defines an abstract `Area` property and derived classes `Rectangle`, `Circle`, and `Triangle`. Unfortunately, for your current application, you don't need the area of a shape; you need its perimeter. You may not be able to modify `Shape` to add a `Perimeter` property (you may not have any control over its source at all), but you know how to compute it for all the classes you're interested in. Before C# 7, a `Perimeter` method might look something like the following listing.

Listing 12.10 Computing a perimeter without patterns

```
static double Perimeter(Shape shape)
{
    if (shape == null)
        throw new ArgumentNullException(nameof(shape));
    Rectangle rect = shape as Rectangle;
    if (rect != null)
        return 2 * (rect.Height + rect.Width);
    Circle circle = shape as Circle;
    if (circle != null)
        return 2 * PI * circle.Radius;
    Triangle triangle = shape as Triangle;
    if (triangle != null)
        return triangle.SideA + triangle.SideB + triangle.SideC;
    throw new ArgumentException(
        $"Shape type {shape.GetType()} perimeter unknown", nameof(shape));
}
```

NOTE If the lack of curly braces inside offends you, I apologize. I normally use them for all loops, if statements, and so forth, but in this case, they ended up dwarfing the useful code here and in some other later pattern examples. I've removed them for brevity.

That's ugly. It's repetitive and long-winded; the same pattern of “check whether the shape is a particular type, and then use that type's properties” occurs three times. Urgh. Importantly, even though there are multiple `if` statements here, the body of each of them returns a value, so you're always picking only one of them to execute. The following listing shows how the same code can be written in C# 7 using patterns in a `switch` statement.

Listing 12.11 Computing a perimeter with patterns

```
static double Perimeter(Shape shape)
{
    switch (shape)
    {
        case null:
            throw new ArgumentNullException(nameof(shape));
        case Rectangle rect:
            return 2 * (rect.Height + rect.Width);
        case Circle circle:
            return 2 * PI * circle.Radius;
        case Triangle tri:
            return tri.SideA + tri.SideB + tri.SideC;
        default:
            throw new ArgumentException(...);
    }
}
```

Handles a null value

Handles each type you know about

If you don't know what to do, throw an exception.

This is quite a departure from the `switch` statement from previous versions of C#, in which case labels were all just constant values. Here you're sometimes interested in

just value matching (for the null case) and sometimes interested in the type of the value (the rectangle, circle, and triangle cases). When you match by type, that match also introduces a new variable of that type that you use to calculate the perimeter.

The topic of patterns within C# has two distinct aspects:

- The syntax for patterns
- The contexts in which you can use patterns

At first, it may feel like everything's new, and differentiating between these two aspects may seem pointless. But the patterns you can use in C# 7.0 are just the start: the C# design team has been clear that the syntax has been designed for new patterns to become available over time. When you know the places in the language where patterns are allowed, you can pick up new patterns easily. It's a little bit chicken and egg—it's hard to demonstrate one part without showing the other—but we'll start by looking at the kinds of patterns available in C# 7.0.

12.4 Patterns available in C# 7.0

C# 7.0 introduces three kinds of patterns: constant patterns, type patterns, and the var pattern. I'm going to demonstrate each with the `is` operator, which is one of the contexts for using patterns.

Every pattern tries to match an input. This can be any nonpointer expression. For the sake of simplicity, I'll refer to this as `input` in the pattern descriptions, as if it were a variable, but it doesn't have to be.

12.4.1 Constant patterns

A *constant pattern* is just what it sounds like: the pattern consists entirely of a compile-time constant expression, which is then checked for equality with `input`. If both `input` and the constant are integer expressions, they're compared using `==`. Otherwise, the static `object.Equals` method is called. It's important that it's the static method that's called, because that enables you to safely check for a null value. The following listing shows an example that serves even less real-world purpose than most of the other examples in the book, but it does demonstrate a couple of interesting points.

Listing 12.12 Simple constant matches

```
static void Match(object input)
{
    if (input is "hello")
        Console.WriteLine("Input is string hello");
    else if (input is 5L)
        Console.WriteLine("Input is long 5");
    else if (input is 10)
        Console.WriteLine("Input is int 10");
    else
        Console.WriteLine("Input didn't match hello, long 5 or int 10");
}
```

```
static void Main()
{
    Match("hello");
    Match(5L);
    Match(7);
    Match(10);
    Match(10L);
}
```

The output is mostly straightforward, but you may be surprised by the penultimate line:

```
Input is string hello
Input is long 5
Input didn't match hello, long 5 or int 10
Input is int 10
Input didn't match hello, long 5 or int 10
```

If integers are compared using `==`, why didn't the last call of `Match(10L)` match? The answer is that the compile-time type of `input` isn't an integral type, it's just `object`, so the compiler generates code equivalent to calling `object.Equals(x, 10)`. That returns `false` when the value of `x` is a boxed `Int64` instead of a boxed `Int32`, as is the case in our last call to `Match`. For an example using `==`, you'd need something like this:

```
long x = 10L;
if (x is 10)
{
    Console.WriteLine("x is 10");
}
```

This isn't useful in `is` expressions like this; it'd be more likely to be used in `switch`, where you might have some integer constants (like a pre-pattern-matching `switch` statement) along with other patterns. A more obviously useful kind of pattern is the type pattern.

12.4.2 Type patterns

A *type pattern* consists of a type and an identifier—a bit like a variable declaration. The pattern matches if `input` is a value of that type, just like the regular `is` operator. The benefit of using a pattern for this is that it also introduces a new *pattern variable* of that type initialized with the value if the pattern matches. If the pattern doesn't match, the variable still exists; it's just not definitely assigned. If `input` is null, it won't match any type. As described in section 12.1.1, the underscore identifier `_` can be used, in which case it's a *discard* and no variable is introduced. The following listing is a conversion of our earlier set of `as`-followed-by-`if` statements (listing 12.10) to use pattern matching without taking the more extreme step of using a `switch` statement.

Listing 12.13 Using type patterns instead of `as`/`if`

```
static double Perimeter(Shape shape)
{
    if (shape == null)
        throw new ArgumentNullException(nameof(shape));
```

```

if (shape is Rectangle rect)
    return 2 * (rect.Height + rect.Width);
if (shape is Circle circle)
    return 2 * PI * circle.Radius;
if (shape is Triangle triangle)
    return triangle.SideA + triangle.SideB + triangle.SideC;
throw new ArgumentException(
    $"Shape type {shape.GetType()} perimeter unknown", nameof(shape));
}

```

In this case, I definitely prefer the `switch` statement option instead, but that would be overkill if you had only one `as/if` to replace. A type pattern is generally used to replace either an `as/if` combination or `if` with `is` followed by a cast. The latter is required when the type you're testing is a non-nullable value type.

The type specified in a type pattern can't be a nullable value type, but it can be a type parameter, and that type parameter may end up being a nullable value type at execution time. In that case, the pattern will match only when the value is non-null. The following listing shows this using `int?` as a type argument for a method that uses the type parameter in a type pattern, even though the expression value `is int? t` wouldn't have compiled.

Listing 12.14 Behavior of nullable value types in type patterns

```

static void Main()
{
    CheckType<int?>(null);
    CheckType<int?>(5);
    CheckType<int?>("text");
    CheckType<string>(null);
    CheckType<string>(5);
    CheckType<string>("text");
}

static void CheckType<T>(object value)
{
    if (value is T t)
    {
        Console.WriteLine($"Yes! {t} is a {typeof(T)}");
    }
    else
    {
        Console.WriteLine($"No! {value ?? "null"} is not a {typeof(T)}");
    }
}

```

The output is as follows:

```

No! null is not a System.Nullable`1[System.Int32]
Yes! 5 is a System.Nullable`1[System.Int32]
No! text is not a System.Nullable`1[System.Int32]
No! null is not a System.String
No! 5 is not a System.String
Yes! text is a System.String

```

To wrap up this section on type patterns, there's one issue in C# 7.0 that's addressed by C# 7.1. It's one of those cases where if your project is already set to use C# 7.1 or higher, you may not even notice. I've included this mostly so that you don't get confused if you copy code from a C# 7.1 project to a C# 7.0 project and find it breaks.

In C# 7.0, type patterns like this

```
x is SomeType y
```

required that the compile-time type of `x` could be cast to `SomeType`. That sounds entirely reasonable until you start using generics. Consider the following generic method that displays details of the shapes provided using pattern matching.

Listing 12.15 Generic method using type patterns

```
static void DisplayShapes<T>(List<T> shapes) where T : Shape
{
    foreach (T shape in shapes)
    {
        switch (shape)
        {
            case Circle c:
                Console.WriteLine($"Circle radius {c.Radius}");
                break;
            case Rectangle r:
                Console.WriteLine($"Rectangle {r.Width} x {r.Height}");
                break;
            case Triangle t:
                Console.WriteLine(
                    $"Triangle sides {t.SideA}, {t.SideB}, {t.SideC}");
                break;
        }
    }
}
```

In C# 7.0, this listing won't compile, because this wouldn't compile either:

```
if (shape is Circle)
{
    Circle c = (Circle) shape;
}
```

The use of the `is` operator is valid, but the cast isn't. The inability to cast type parameters directly has been an annoyance for a long time in C#, with the usual workaround being to first cast to `object`:

```
if (shape is Circle)
{
    Circle c = (Circle) (object) shape;
}
```

This is clumsy enough in a normal cast, but it's worse when you're trying to use an elegant type pattern.

In listing 12.15, this can be worked around by either accepting an `IEnumerable<Shape>` (taking advantage of generic covariance to allow a conversion of `List<Circle>` to `IEnumerable<Shape>`, for example) or by specifying the type of shape as `Shape` instead of `T`. In other cases, the workarounds aren't as simple. C# 7.1 addresses this by permitting a type pattern for any type that would be valid using the `as` operator, which makes listing 12.15 valid.

I expect the type pattern to be the most commonly used pattern out of the three patterns introduced in C# 7.0. Our final pattern almost doesn't sound like a pattern at all.

12.4.3 The var pattern

The `var` pattern looks like a type pattern but using `var` as the type, so it's just `var` followed by an identifier:

```
someExpression is var x
```

Like type patterns, it introduces a new variable. But unlike type patterns, it doesn't test anything. It always matches, resulting in a new variable with the same compile-time type as `input`, with the same value as `input`. Unlike type patterns, the `var` pattern still matches even if `input` is a null reference.

Because it always matches, using the `var` pattern with the `is` operator in an `if` statement in the way that I've demonstrated for the other patterns is reasonably pointless. It's most useful with `switch` statements in conjunction with a *guard clause* (described in section 12.6.1), although it could also occasionally be useful if you want to switch on a more complex expression without assigning it to a variable.

Just for the sake of presenting an example of `var` without using guard clauses, listing 12.16 shows a `Perimeter` method similar to the one in listing 12.11. But this time, if the `shape` parameter has a null value, a random shape is created instead. You use a `var` pattern to report the type of the shape if you then can't compute the perimeter. You don't need the constant pattern with the value `null` now, as you're ensuring that you never switch on a null reference.

Listing 12.16 Using the var pattern to introduce a variable on error

```
static double Perimeter(Shape shape)
{
    switch (shape ?? CreateRandomShape())
    {
        case Rectangle rect:
            return 2 * (rect.Height + rect.Width);
        case Circle circle:
            return 2 * PI * circle.Radius;
        case Triangle triangle:
            return triangle.SideA + triangle.SideB + triangle.SideC;
        case var actualShape:
            throw new InvalidOperationException(
                $"Shape type {actualShape.GetType()} perimeter unknown");
    }
}
```

In this case, an alternative would've been to introduce the `actualShape` variable before the `switch` statement, switch on that, and then use the default case as before.

Those are all the patterns available in C# 7.0. You've already seen both of the contexts in which they can be used—with the `is` operator and in `switch` statements—but there's a little more to say in each case.

12.5 Using patterns with the `is` operator

The `is` operator can be used anywhere as part of a normal expression. It's almost always used with `if` statements, but it certainly doesn't have to be. Until C# 7, the right-hand side of an `is` operator had to be just a type, but now it can be any pattern. Although this does allow you to use the constant or `var` patterns, realistically you'll almost always use type patterns instead.

Both the `var` pattern and type patterns introduce a new variable. Prior to C# 7.3, this came with an extra restriction: you can't use them in field, property, or constructor initializers or query expressions. For example, this would be invalid:

```
static int length = GetObject() is string text ? text.Length : -1;
```

I haven't found this to be an issue, but the restriction is lifted in C# 7.3 anyway.

That leaves us with patterns introducing local variables, which leads to an obvious question: what's the scope of the newly introduced variable? I understand that this was the cause of a lot of discussion within the C# language team and the community, but the final result is that the scope of the introduced variable is the enclosing block.

As you might expect from a hotly debated topic, there are pros and cons to this. One of the things I've never liked about the `as/if` pattern shown in listing 12.10 is that you end up with a lot of variables in scope even though you typically don't want to use them outside the condition where the value matched the type you were testing. Unfortunately, this is still the case when using type patterns. It's not quite the same situation, as the variable won't be definitely assigned in branches where the pattern wasn't matched.

To compare, after this code

```
string text = input as string;
if (text != null)
{
    Console.WriteLine(text);
}
```

the `text` variable is in scope and definitely assigned. The roughly equivalent type pattern code looks like this:

```
if (input is string text)
{
    Console.WriteLine(text);
}
```

After this, the `text` variable is in scope, but not definitely assigned. Although this does pollute the declaration space, it can be useful if you're trying to provide an alternative way of obtaining a value. For example:

```
if (input is string text)
{
    Console.WriteLine("Input was already a string; using that");
}
else if (input is StringBuilder builder)
{
    Console.WriteLine("Input was a StringBuilder; using that");
    text = builder.ToString();
}
else
{
    Console.WriteLine(
        $"Unable to use value of type ${input.GetType()}. Enter text:");
    text = Console.ReadLine();
}
Console.WriteLine($"Final result: {text}");
```

Here you really want the `text` variable to stay in scope, because you want to use it; you assign to it in one of two ways. You don't really want `builder` in scope after the middle block, but you can't have it both ways.

To be a little more technical about the definite assignment, after an `is` expression with a pattern that introduces a pattern variable, the variable is (in language specification terminology) “definitely assigned after true expression.” That can be important if you want an `if` condition to do more than just test the type. For example, suppose you want to check whether the value provided is a large integer. This is fine:

```
if (input is int x && x > 100)
{
    Console.WriteLine($"Input was a large integer: {x}");
}
```

You can use `x` after the `&&` because you'll evaluate that operand only if the first operand evaluates to `true`. You can also use `x` inside the `if` statement because you'll execute the body of the `if` statement only if both `&&` operands evaluate to `true`. But what if you want to handle both `int` or `long` values? You can test the value, but then you can't tell which condition matched:

```
if ((input is int x && x > 100) || (input is long y && y > 100))
{
    Console.WriteLine($"Input was a large integer of some kind");
}
```

Here, both `x` and `y` are in scope both inside and after the `if` statement, even though the part declaring `y` looks as if it may not execute. But the variables are definitely assigned only within the very small piece of code where you're checking how large the values are.

All of this makes logical sense, but it can be a little surprising the first time you see it. The two takeaways of this section are as follows:

- Expect the scope of a pattern variable declared in an `is` expression to be the enclosing block.
- If the compiler prevents you from using a pattern variable, that means the language rules can't prove that the variable will have been assigned a value at that point.

In the final part of this chapter, we'll look at patterns used in `switch` statements.

12.6 Using patterns with switch statements

Specifications are often written not in terms of algorithms as such but in terms of cases. The following are examples far removed from computing:

- *Taxes and benefits*—Your tax bracket probably depends on your income and some other factors.
- *Travel tickets*—There may be group discounts as well as separate prices for children, adults, and the elderly.
- *Takeout food ordering*—There can be deals if your order meets certain criteria.

In the past, we've had two ways of detecting which case applies to a particular input: `switch` statements and `if` statements, where `switch` statements were limited to simple constants. We still have just those two approaches, but `if` statements are already cleaner using patterns as you've seen, and `switch` statements are much more powerful.

NOTE Pattern-based `switch` statements feel quite different from the constant-value-only `switch` statements of the past. Unless you've had experience with other languages that have similar functionality, you should expect it to take a little while to get used to the change.

`switch` statements with patterns are largely equivalent to a sequence of `if/else` statements, but they encourage you to think more in terms of “this kind of input leads to this kind of output” instead of steps.

All switch statements can be considered pattern based

Throughout this section, I talk about constant-based `switch` statements and pattern-based `switch` statements as if they're different. Because constant patterns are patterns, every valid `switch` statement can be considered a pattern-based `switch` statement, and it will still behave in exactly the same way. The differences you'll see later in terms of execution order and new variables being introduced don't apply to constant patterns anyway.

I find it quite helpful, at least at the moment, to consider these as if they were two separate constructs that happen to use the same syntax. You may feel more comfortable not to make that distinction. It's safe to use either mental model; they'll both predict the code's behavior correctly.

You've already seen an example of patterns in `switch` statements in section 12.3, where you used a constant pattern to match `null` and type patterns to match different kinds of shapes. In addition to simply putting a pattern in the case label, there's one new piece of syntax to introduce.

12.6.1 Guard clauses

Each case label can also have a guard clause, which consists of an expression:

```
case pattern when expression:
```

The expression has to evaluate to a Boolean value² just like an `if` statement's condition. The body of the case will be executed only if the expression evaluates to `true`. The expression can use more patterns, thereby introducing extra pattern variables.

Let's look at a concrete example that'll also illustrate my point about specifications. Consider the following definition of the Fibonacci sequence:

- `fib(0) = 0`
- `fib(1) = 1`
- `fib(n) = fib(n-2) + fib(n-1)` for all $n > 1$

In chapter 11, you saw how to generate the Fibonacci sequence by using tuples, which is a clean approach when considering it as a sequence. If you consider it only as a function, however, the preceding definition leads to the following listing: a simple `switch` statement using patterns and a guard clause.

Listing 12.17 Implementing the Fibonacci sequence recursively with patterns

```
static int Fib(int n)
{
    switch (n)
    {
        case 0: return 0;
        case 1: return 1;
        case var _ when n > 1: return Fib(n - 2) + Fib(n - 1);
        default: throw new ArgumentOutOfRangeException(
            nameof(n), "Input must be non-negative");
    }
}
```

Base cases handled with constant patterns

Recursive case handled with var pattern and guard clause

If you don't match any patterns, the input was invalid.

This is a horribly inefficient implementation that I'd never use in real life, but it clearly demonstrates how a specification can be directly translated into code.

In this example, the guard clause doesn't need to use the pattern variable, so I used a discard with the `_` identifier. In many cases, if the pattern introduces a new variable, it *will* be used in the guard clause or at least in the case body.

When you use guard clauses, it makes perfect sense for the same pattern to appear multiple times, because the first time the pattern matches, the guard clause

² It can also be a value that can be implicitly converted to a Boolean value or a value of a type that provides a `true` operator. These are the same requirements as the condition in an `if` statement.

may evaluate to `false`. Here's an example from Noda Time in a tool used to build documentation:

```
private string GetUid(TypeReference type, bool useTypeArgumentNames)
{
    switch (type)
    {
        case ByReferenceType brt:
            return $"{GetUid(brt.ElementType, useTypeArgumentNames)}@";
        case GenericParameter gp when useTypeArgumentNames:
            return gp.Name;
        case GenericParameter gp when gp.DeclaringType != null:
            return $"`{gp.Position}";
        case GenericParameter gp when gp.DeclaringMethod != null:
            return $"`{gp.Position}";
        case GenericParameter gp:
            throw new InvalidOperationException(
                "Unhandled generic parameter");
        case GenericInstanceType git:
            return "(This part of the real code is long and irrelevant)";
        default:
            return type.FullName.Replace('/', '.');
    }
}
```

I have four patterns that handle generic parameters based on the `useTypeArgumentNames` method parameter and then whether the generic type parameter was introduced in a method or a type. The case that throws an exception is almost a `default` case for generic parameters, indicating that it's come across a situation I haven't thought about yet. The fact that I'm using the same pattern variable name (`gp`) for multiple cases raises another natural question: what's the scope of a pattern variable introduced in a case label?

12.6.2 Pattern variable scope for case labels

If you declare a local variable directly within a case body, the scope of that variable is the whole `switch` statement, including other case bodies. That's still true (and unfortunate, in my opinion), but it doesn't include variables declared within case labels. The scope of those variables is just the body associated with that case label. That applies to pattern variables declared by the pattern, pattern variables declared within the guard clause, and any out variables (see section 14.2) declared in the guard clause.

That's almost certainly what you want, and it's useful in terms of allowing you to use the same pattern variables for multiple cases handling similar situations, as demonstrated in the Noda Time tool code. There's one quirk here: just as with normal `switch` statements, it's valid to have multiple case labels with the same body. At that point, the variables declared within all the case labels for that body are required to have different names (because they're contributing to the same declaration space). But within the case body, none of those variables will be definitely assigned, because the compiler can't tell which label matched. It can still be useful to introduce those variables, but mostly for the sake of using them in guard clauses.

For example, suppose you’re matching an `object` input, and you want to make sure that if it’s numeric, it’s within a particular range, and that range may vary by type. You could use one type pattern per numeric type with a corresponding guard clause. The following listing shows this for `int` and `long`, but you could expand it for other types.

Listing 12.18 Using multiple case labels with patterns for a single case body

```
static void CheckBounds(object input)
{
    switch (input)
    {
        case int x when x > 1000:
        case long y when y > 10000L:
            Console.WriteLine("Value is too large");
            break;
        case int x when x < -1000:
        case long y when y < -10000L:
            Console.WriteLine("Value is too low");
            break;
        default:
            Console.WriteLine("Value is in range");
            break;
    }
}
```

The pattern variables are definitely assigned within the guard clauses, because execution will reach the guard clause only if the pattern has matched to start with, and they’re still in scope within the body, but they’re not definitely assigned. You could assign new values to them and use them after that, but I feel that won’t often be useful.

In addition to the basic premise of pattern matching being new and different, there’s one huge difference between the constant-based `switch` statements of the past and new pattern-based `switch` statements: the order of cases matters in a way that it didn’t before.

12.6.3 Evaluation order of pattern-based switch statements

In almost all situations, case labels for constant-based `switch` statements can be reordered freely with no change in behavior.³ This is because each case label matches a single constant value, and the constants used for any `switch` statement all have to be different, so any input can match at most only one case label. With patterns, that’s no longer true.

The logical evaluation order of a pattern-based `switch` statement can be summarized simply:

³ The only time this isn’t true is when you use a variable in one case body that was declared in an earlier case body. That’s almost always a bad idea anyway, and it’s a problem only because of the shared scope of such variables.

- Each `case` label is evaluated in source-code order.
- The code body of the `default` label is executed only when all the `case` labels have been evaluated, regardless of where the `default` label is within the `switch` statement.

TIP Although you now know that the code associated with the `default` label is executed only if none of the `case` labels matches, regardless of where it appears, it's possible that some people reading your code might not. (Indeed, you might have forgotten it by the time you next come to read your own code.) If you put the `default` label as the final part of the `switch` statement, the behavior is always clear.

Sometimes it won't matter. In our Fibonacci-computing method, for example, the cases were only 0, 1, and more than 1, so they could be freely reordered. Our Noda Time tool code, however, had four cases that definitely need to be checked in order:

```
case GenericParameter gp when useTypeArgumentNames:
    return gp.Name;
case GenericParameter gp when gp.DeclaringType != null:
    return $"^{gp.Position}";
case GenericParameter gp when gp.DeclaringMethod != null:
    return $"~~{gp.Position}";
case GenericParameter gp:
    throw new InvalidOperationException(...);
```

Here you want to use the generic type parameter name whenever `useTypeArgumentNames` is true (the first case), regardless of the other cases. The second and third cases are mutually exclusive (in a way that you know but the compiler wouldn't), so their order doesn't matter. The last case must be last within these four because you want the exception to be thrown only if the input is a `GenericParameter` that isn't otherwise handled.

The compiler is helpful here: the final case doesn't have a guard clause, so it'll always be valid if the type pattern matches. The compiler is aware of this; if you put that case earlier than the other case labels with the same pattern, it knows that's effectively hiding them and reports an error.

Multiple case bodies can be executed in only one way, and that's with the rarely used `goto` statement. That's still valid within pattern-based `switch` statements, but you can `goto` only a constant value, and a `case` label must be associated with that value without a guard clause. For example, you can't `goto` a type pattern, and you can't `goto` a value on the condition that an associated guard clause also evaluates to `true`. In reality, I've seen so few `goto` statements in `switch` statements that I can't see this being much of a restriction.

I deliberately referred to the *logical* evaluation order earlier. Although the C# compiler could effectively translate every `switch` statement into a sequence of `if/else` statements, it can act more efficiently than that. For example, if there are multiple type patterns for the same type but with different guard clauses, it can evaluate the type pattern part once and then check each guard clause in turn instead. Similarly, for constant

values without guard patterns (which still have to be distinct, just as in previous versions of C#), the compiler can use the IL `switch` instruction, potentially after performing an implicit type check. Exactly which optimizations the compiler performs is beyond the scope of this book, but if you ever happen to look at the IL associated with a `switch` statement and it bears little resemblance to the source code, this may well be the cause.

12.7 Thoughts on usage

This section provides preliminary thoughts on how the features described in this chapter are best used. Both features are likely to evolve further and possibly even be combined with a deconstruction pattern. Other related potential features, such as syntax to write an expression-bodied method for which the result is based on a pattern-based `switch`, may well affect where these features are used. You'll see some potential C# 8 features like this in chapter 15.

Pattern matching is an implementation concern, which means that you don't need to worry if you find later that you've overused it. You can revert to an older style of coding if you find patterns don't give you the readability benefit you'd expected. The same is true of deconstruction to some extent. But if you've added public `Deconstruct` methods all over your API, removing them would be a breaking change.

More than that, I suggest that most types aren't naturally deconstructable anyway, just as most types don't have a natural `IComparable<T>` implementation. I suggest adding a `Deconstruct` method only if the order of the components is obvious and unambiguous. That's fine for coordinates, anything with a hierarchical nature such as date/time values, or even where there's a common convention, such as colors being thought of as RGB with optional alpha. Most business-related entities probably don't fall into this category, though; for example, an item in an online shopping basket has various aspects, but there's no obvious order to them.

12.7.1 Spotting deconstruction opportunities

The simplest kind of deconstruction to use is likely to be related to tuples. If you're calling a method that returns a tuple and you don't need to keep the values together, consider deconstructing them instead. For example, with our `MinMax` method from chapter 11, I'd almost always deconstruct immediately instead of keeping the return value as a tuple:

```
int[] values = { 2, 7, 3, -5, 1, 0, 10 };
var (min, max) = MinMax(values);
Console.WriteLine(min);
Console.WriteLine(max);
```

I suspect the use of nontuple deconstruction will be rarer, but if you're dealing with points, colors, date/time values, or something similar, you may find that it's worth deconstructing the value early on if you'd otherwise refer to the components via properties multiple times. You could've done this before C# 7, but the ease of declaring multiple local variables via deconstruction could easily swing the balance between not worth doing and worth doing.

12.7.2 Spotting pattern matching opportunities

You should consider pattern matching in two obvious places:

- Anywhere you’re using the `is` or `as` operators and conditionally executing code by using the more specifically typed value.
- Anywhere you have an `if/else-if/else-if/else` sequence using the same value for all the conditions, and you can use a `switch` statement instead.

If you find yourself using a pattern of the form `var ...` when multiple times (in other words, when the only condition occurs in a guard clause), you may want to ask yourself whether this is really pattern matching. I’ve certainly come across scenarios like that, and so far I’ve erred on the side of using pattern matching anyway. Even if it feels slightly abusive, it conveys the intent of matching a single condition and taking a single action more clearly than the `if/else` sequence does, in my view.

Both of these are transformations of an existing code structure with changes only to the implementation details. They’re not changing the way you think about and organize your logic. That grander style of change—which could still be refactoring within the visible API of a single type, or perhaps within the public API of an assembly, by changing internal details—is harder to spot. Sometimes it may be a move away from using inheritance; the logic for a calculation may be more clearly expressed in a single place that considers all the different cases than as part of the type representing each of those cases. The perimeter of a shape case in section 12.3 is one example of this, but you could easily apply the same ideas to many business cases. This is where disjoint union types are likely to become more widespread within C#.

As I said, these are preliminary thoughts. As always, I encourage you to experiment with deliberate introspection: consider opportunities as you code, and if you try something new, reflect on its pros and cons after you’ve done so.

Summary

- Deconstruction allows you to break values into multiple variables with syntax that’s consistent between tuples and nontuples.
- Nontuple types are deconstructed using a `Deconstruct` method with `out` parameters. This can be an extension method or an instance method.
- Multiple variables can be declared with a single `var` deconstruction if all the types can be inferred by the compiler.
- Pattern matching allows you to test the type and content of a value, and some patterns allow you to declare a new variable.
- Pattern matching can be used with the `is` operator or in `switch` statements.
- A pattern within a `switch` statement can have an additional guard clause introduced by the `when` contextual keyword.
- When a `switch` statement contains patterns, the order of the `case` labels can change the behavior.

13

Improving efficiency with more pass by reference

This chapter covers

- Aliasing variables with the `ref` keyword
- Returning variables by reference with `ref` returns
- Efficient argument passing with `in` parameters
- Preventing data changes with read-only `ref` returns, read-only `ref` locals, and read-only struct declarations
- Extension methods with `in` or `ref` targets
- `Ref-like` structs and `Span<T>`

When C# 7.0 came out, it had a couple of features that struck me as slightly odd: `ref` local variables and `ref` returns. I was slightly skeptical about how many developers would need them, as they seemed to be targeted situations involving large value types, which are rare. My expectation was that only near-real-time services and games would find these useful.

C# 7.2 brought another raft of `ref`-related features: `in` parameters, read-only `ref` locals and returns, read-only structs, and `ref-like` structs. These were complementary

to the 7.0 features but still appeared to be making the language more complicated for the benefit of a small set of users.

I'm now convinced that although many developers may not directly see more ref-based code in their projects, they'll reap the benefits of the features existing because more-efficient facilities are being made available in the framework. At the time of writing, it's too early to say for sure how revolutionary this will prove, but I think it's likely to be significant.

Often performance comes at the expense of readability. I still believe that's the case with many of the features described in this chapter; I'm expecting them to be used sparingly in cases where performance is known to be important enough to justify the cost. The framework changes enabled by all of this are a different matter, though. They should make it reasonably easy to reduce object allocations and save both memory and garbage collector work without making your code harder to read.

I bring all of this up because you may have similar reactions. While reading this chapter, it's entirely reasonable to decide that you'll try to avoid most of the language features here. I urge you to plow on to the end, though, to see the framework-related benefits. The final section, on ref-like structs, introduces `Span<T>`. Far more can be said about spans than I have room to write in this book, but I expect spans and related types to be important parts of the developer toolbox in the future.

Throughout this chapter, I'll mention when a feature is available only in a point release of C# 7. As with other point release features, that means if you're using a C# 7 compiler, you'll be able to take advantage of those features only with appropriate project settings to specify the language version. I suggest you take an all-or-nothing approach to ref-related features: either use them all, with appropriate settings to allow this, or use none of them. Using only the features in C# 7.0 is likely to be less satisfying. With all of that said, let's start by revisiting the use of the `ref` keyword in earlier versions of C#.

13.1 Recap: What do you know about `ref`?

You need a firm grasp of how `ref` parameters work in C# 6 and earlier in order to understand the `ref`-related features in C# 7. This, in turn, requires a firm grasp of the difference between a variable and its value.

Different developers have different ways of thinking about variables, but my mental model is always that of a piece of paper, as shown in figure 13.1. The piece of paper has three items of information:

- The name of the variable
- The compile-time type
- The current value

Assigning a new value to the variable is just a matter of erasing the current value and writing a new one instead.

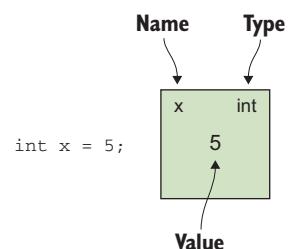


Figure 13.1 Representing a variable as a piece of paper

When the type of the variable is a reference type, the value written on the piece of paper is never an object; it's always an object reference. An object reference is just a way of navigating to an object in the same way that a street address is a way of navigating to a building. Two pieces of paper with the same address written on them refer to the same building, just as two variables with the same reference value refer to the same object.

TIP The `ref` keyword and object references are different concepts. Similarities certainly exist, but you need to distinguish between them. Passing an object reference by value isn't the same thing as passing a variable by reference, for example. In this section, I've emphasized the difference by using *object reference* instead of just *reference*.

Importantly, when an assignment copies one variable's value into another variable, it really is just the value that's copied; the two pieces of paper remain independent, and a later change to either variable doesn't change the other. Figure 13.2 illustrates this concept.

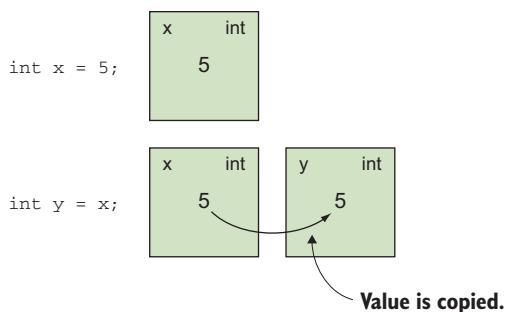


Figure 13.2 Assignment copying a value into a new variable

This sort of value copying is exactly what happens with a value parameter when you call a method; the *value* of the method argument is copied onto a fresh piece of paper—the parameter—as shown in figure 13.3. The argument doesn't have to be a variable; it can be any expression of an appropriate type.

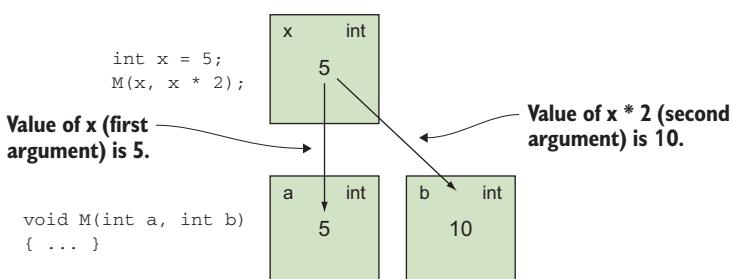


Figure 13.3 Calling a method with value parameters: the parameters are new variables that start with the values of the arguments.

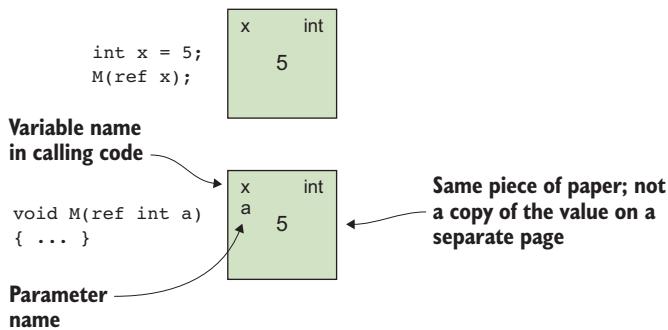


Figure 13.4 A `ref` parameter uses the same piece of paper rather than creating a new one with a copy of the value.

A `ref` parameter behaves differently, as shown in figure 13.4. Instead of acting as a new piece of paper, a reference parameter requires the caller to provide an existing piece of paper, not just an initial value. You can think of it as a piece of paper with two names written on it: the one the calling code uses to identify it and the parameter name.

If the method modifies the value of the `ref` parameter, thereby changing what's written on the paper, then when the method returns, that change is visible to the caller because it's on the original piece of paper.

NOTE There are different ways of thinking about `ref` parameters and variables. You may read other authors who treat `ref` parameters as entirely separate variables that just have an automatic layer of indirection so that any access to the `ref` parameter follows the indirection first. That's closer to what the IL represents, but I find it less helpful.

There's no requirement that each `ref` parameter uses a different piece of paper. The following listing provides a somewhat extreme example, but it's good for checking your understanding before moving on to `ref` locals.

Listing 13.1 Using the same variable for multiple `ref` parameters

```
static void Main()
{
    int x = 5;
    IncrementAndDouble(ref x, ref x);
    Console.WriteLine(x);
}

static void IncrementAndDouble(ref int p1, ref int p2)
{
    p1++;
    p2 *= 2;
}
```

The output here is 12: `x`, `p1`, `p2` all represent the same piece of paper. It starts with a value of 5; `p1++` increments it to 6, and `p2 *= 2` doubles it to 12. Figure 13.5 shows a graphical representation of the variables involved.

A common way of talking about this is *aliasing*: in the preceding example, the variables `x`, `p1`, and `p2` are all *aliases* for the same storage location. They're different ways of getting to the same piece of memory.

Apologies if this seems long-winded and old hat. You're now ready to move on to the genuinely new features of C# 7. With the mental model of variables as pieces of paper, understanding the new features will be much easier.

13.2 Ref locals and ref returns

Many of the ref-related C# 7 features are interconnected, which makes it harder to understand the benefits when you see them one at a time. While I'm describing the features, the examples will be even more contrived than normal, as they try to demonstrate just a single point at a time. The first two features you'll look at are the ones introduced in C# 7.0, although even they were enhanced in C# 7.2. First up, ref locals.

13.2.1 Ref locals

Let's continue our earlier analogy: ref parameters allow a piece of paper to be shared between variables in two methods. The same piece of paper used by the caller is the one that the method uses for the parameter. Ref locals take that idea one step further by allowing you declare a new local variable that shares the same piece of paper as an existing variable.

The following listing shows a trivial example of this, incrementing twice via different variables and then showing the result. Note that you have to use the `ref` keyword in both the declaration and in the initializer.

Listing 13.2 Incrementing twice via two variables

```
int x = 10;
ref int y = ref x;
x++;
y++;
Console.WriteLine(x);
```

This prints 12, just as if you'd incremented `x` twice.

Any expression of the appropriate type that's classified as a variable can be used to initialize a ref local, including array elements. If you have an array of large mutable value types, this can avoid unnecessary copy operations in order to make multiple

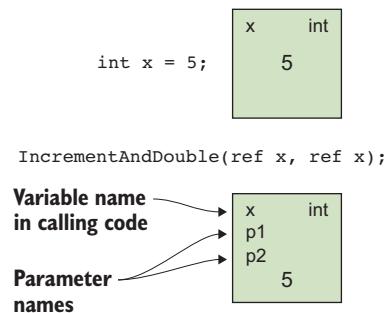


Figure 13.5 Two ref parameters referring to the same piece of paper

changes. The following listing creates an array of tuples and then modifies both items within each array element without copying.

Listing 13.3 Modifying array elements using ref local

```
var array = new (int x, int y)[10];

for (int i = 0; i < array.Length; i++)
{
    array[i] = (i, i);
```

Initializes the array with
(0, 0), (1, 1), and so on

```
for (int i = 0; i < array.Length; i++)
{
    ref var element = ref array[i];
    element.x++;
    element.y *= 2;
}
```

For each element of
the array, increments x
and doubles y

Before ref locals, there would've been two alternatives to modify the array. You could use either multiple array access expressions such as the following:

```
for (int i = 0; i < array.Length; i++)
{
    array[i].x++;
    array[i].y *= 2;
}
```

Or you could copy the whole tuple out of the array, modify it, and then copy it back:

```
for (int i = 0; i < array.Length; i++)
{
    var tuple = array[i];
    tuple.x++;
    tuple.y *= 2;
    array[i] = tuple;
}
```

Neither of these is particularly appealing. The ref local approach expresses our aim of working with an array element as a normal variable for the body of the loop.

Ref locals can also be used with fields. The behavior for a static field is predictable, but the behavior for instance fields may surprise you. Consider the following listing, which creates a ref local to alias a field in one instance via a variable (`obj`) and then changes the value of `obj` to refer to a different instance.

Listing 13.4 Aliasing the field of a specific object by using ref local

```
class RefLocalField
{
    private int value;

    static void Main()
    {
```

```

    var obj = new RefLocalField();
    ref int tmp = ref obj.value;
    tmp = 10;
    Console.WriteLine(obj.value);

    obj = new RefLocalField();
    Console.WriteLine(tmp);
    Console.WriteLine(obj.value);
}

} // Demonstrates that the value of the
   // field in the second instance really is 0
  
```

Assigns a new value to ref local

Demonstrates that this has modified the field

Creates an instance of RefLocalField

Declares a ref local variable referring to the field of the first instance

Reassigns the obj variable to refer to a second instance of RefLocalField

Demonstrates that tmp still uses the field in the first instance

The output is shown here:

```

10
10
0
  
```

The possibly surprising line is the middle one. It demonstrates that using `tmp` isn't the same as using `obj.value` each time. Instead, `tmp` acts as an alias for the field expressed as `obj.value` at the point of initialization. Figure 13.6 shows a snapshot of the variables and objects involved at the end of the `Main` method.

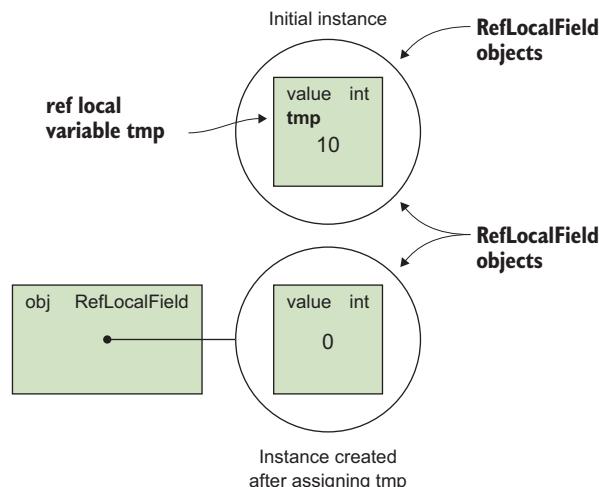


Figure 13.6 At the end of listing 13.4, the `tmp` variable refers to a field in the first instance created, whereas the value of `obj` refers to a different instance.

As a corollary of this, the `tmp` variable will prevent the first instance from being garbage collected until after the last use of `tmp` in the method. Similarly, using a ref local for an array element stops the array containing that element from being garbage collected.

NOTE A ref variable that refers to a field within an object or an array element makes life harder for the garbage collector. It has to work out which object the variable is part of and keep that object alive. Regular object references are simpler because they directly identify the object involved. Each ref variable that refers to a field in an object introduces an *interior pointer* into a data structure

maintained by the garbage collector. It'd be expensive to have a lot of these present concurrently, but ref variables can occur only on the stack, which makes it less likely that there'll be enough to cause performance issues.

Ref locals do have a few restrictions around their use. Most of them are obvious and won't get in your way, but it's worth knowing them just so you don't experiment to try to work around them.

INITIALIZATION: ONCE, ONLY ONCE, AND AT DECLARATION (BEFORE C# 7.3)

Ref locals always have to be initialized at the point of declaration. For example, the following code is invalid:

```
int x = 10;
ref int invalid;
invalid = ref int x;
```

Likewise, there's no way to change a ref local to alias a different variable. (In our model terms, you can't rub the name off and then write it on a different piece of paper.) Of course, the same variable can effectively be declared several times; for example, in listing 13.3, you declared the element variable in a loop:

```
for (int i = 0; i < array.Length; i++)
{
    ref var element = ref array[i];
    ...
}
```

On each iteration of the loop, `element` will alias a different array element. But that's okay, because it's effectively a new variable on each iteration.

The variable used to initialize the ref local has to be definitely assigned, too. You might expect the variables to share definite assignment status, but rather than making the definite assignment rules even more complicated, the language designers ensured that ref locals are always definitely assigned. Here's an example:

```
int x;
ref int y = ref x;           ← Invalid, as x isn't
x = 10;                     | definitely assigned
Console.WriteLine(y);
```

This code doesn't try to read from any variable until everything is definitely assigned, but it's still invalid.

C# 7.3 lifts the restriction on reassignment, but ref locals still have to be initialized at the point of declaration using a definitely assigned variable. For example:

```
int x = 10;
int y = 20;
ref int r = ref x;
r++;
r = ref y;                 ← Valid only
                           | in C# 7.3
r++;
Console.WriteLine($"x={x}; y={y}");   ← Prints x = 11,
                                         | y = 21
```

I urge a degree of caution around using this feature anyway. If you need the same ref variable to refer to different variables over the course of a method, I suggest at least trying to refactor the method to be simpler.

NO REF FIELDS, OR LOCAL VARIABLES THAT WOULD LIVE BEYOND THE METHOD CALL

Although a ref local can be initialized using a field, you can't declare a field using `ref`. This is one aspect of protecting against having a ref variable that acts like an alias for another variable with a shorter lifetime. It'd be problematic if you could create an object with a field that aliased a local variable in a method; what would happen to that field after the method had returned?

The same concern around lifetimes extends to local variables in three cases:

- Iterator blocks can't contain ref locals.
- Async methods can't contain ref locals.
- Ref locals can't be captured by anonymous methods or local methods. (Local methods are described in chapter 14.)

These are all cases where local variables can live beyond the original method call. At times, the compiler could potentially prove that it wouldn't cause a problem, but the language rules have been chosen for simplicity. (One simple example of this is a local method that's only called by the containing method rather than being used in a method group conversion.)

NO REFERENCES TO READ-ONLY VARIABLES

Any ref local variable introduced in C# 7.0 is writable; you can write a new value on the piece of paper. That causes a problem if you try to initialize the ref local by using a piece of paper that isn't writable. Consider this attempt to violate the `readonly` modifier:

```
class MixedVariables
{
    private int writableField;
    private readonly int readonlyField;

    public void TryIncrementBoth()
    {
        ref int x = ref writableField;
        ref int y = ref readonlyField; ← Aliases a
                                       ← Attempts to alias
                                       a writable field
                                       a readonly field
        x++;      | Increments both variables
        y++;      |
    }
}
```

If this were valid, all the reasoning we've built up over the years about read-only fields would be lost. Fortunately, that isn't the case; the compiler prevents the assignment to `y` just as it would prevent any direct modification of `readonlyField`. But this code would be valid in the constructor for the `MixedVariables` class, because in that situation you'd be able to write directly to `readonlyField` as well. In short, you can initialize a ref local only in a way that aliases a variable you'd be able to write to in other

situations. This matches the behavior from C# 1.0 onward for using fields as arguments for ref parameters.

This restriction can be frustrating if you want to take advantage of the sharing aspect of ref locals without the writable aspect. In C# 7.0, that's a problem; but you'll see in section 13.2.4 that C# 7.2 offers a solution.

TYPES: ONLY IDENTITY CONVERSIONS ARE PERMITTED

The type of the ref local either has to be the same as the type of the variable it's being initialized with or there has to be an identity conversion between the two types. Any other conversion—even reference conversions that are allowed in many other scenarios—areen't enough. The following listing shows an example of a ref local declaration using a tuple-based identity conversion that you learned about in chapter 11.

NOTE See section 11.3.3 for a reminder on identity conversions.

Listing 13.5 Identity conversion in ref local declaration

```
(int x, int y) tuple1 = (10, 20);
ref (int a, int b) tuple2 = ref tuple1;
tuple2.a = 30;
Console.WriteLine(tuple1.x);
```

This prints 30, as tuple1 and tuple2 share the same storage location; tuple1.x and tuple2.a are equivalent to each other, as are tuple1.y and tuple2.b.

In this section, you've looked at initializing ref locals from local variables, fields, and array elements. A new kind of expression is categorized as a variable in C# 7: the variable returned by a ref return method.

13.2.2 Ref returns

In some ways, it should be easy to understand ref returns. Using our previous model, it's the idea that a method can return a piece of paper instead of a value. You need to add the `ref` keyword to the return type and to any return statement. The calling code will often declare a ref local to receive the return value, too. This means you have to sprinkle the `ref` keyword pretty liberally in your code to make it very clear what you're trying to do. The following listing shows about the simplest possible use of ref return; the `RefReturn` method returns whatever variable was passed into it.

Listing 13.6 Simplest possible ref return demonstration

```
static void Main()
{
    int x = 10;
    ref int y = ref RefReturn(ref x);
    y++;
    Console.WriteLine(x);
}
```

```
static ref int RefReturn(ref int p)
{
    return ref p;
}
```

This prints 11, because x and y are on the same piece of paper, just as if you'd written

```
ref int y = ref x;
```

The method is essentially an identity function just to show the syntax. It could've been written as an expression-bodied method, but I wanted to make the return part clear.

So far, so simple, but a lot of details get in the way, mostly because the compiler makes sure that any piece of paper that's returned is still going to exist when the method has finished returning. It can't be a piece of paper that was created in the method.

To put this in implementation terms, a method can't return a storage location that it's just created on the stack, because when the stack is popped, the storage location won't be valid anymore. When describing how the C# language works, Eric Lippert is fond of saying that the stack is an implementation detail (see <http://mng.bz/oVvZ>). In this case, it's an implementation detail that leaks into the language. The restrictions are for the same reasons that ref fields are prohibited, so if you feel you understand one of these, you can apply the same logic to the other.

I won't go into an exhaustive list of every kind of variable that can and can't be returned using ref return, but here are the most common examples:

VALID

- **ref** or **out** parameters
- Fields of reference types
- Fields of structs where the struct variable is a **ref** or **out** parameter
- Array elements

INVALID

- Local variables declared in the method (including value parameters)
- Fields of struct variables declared in the method

In addition to these restrictions on what can and can't be returned, ref return is entirely invalid in async methods and iterator blocks. Similar to pointer types, you can't use the **ref** modifier in a type argument, although it can appear in interface and delegate declarations. For example, this is entirely valid

```
delegate ref int RefFuncInt32();
```

But you couldn't get the same result by trying to refer to `Func<ref int>`.

Ref return doesn't have to be used with a ref local. If you want to perform a single operation on the result, you can do that directly. The following listing shows this using the same code as listing 13.6 but without the ref local.

Listing 13.7 Incrementing the result of a ref return directly

```
static void Main()
{
    int x = 10;
    RefReturn(ref x)++;           ← Increments the returned
    Console.WriteLine(x);         variable directly
}

static ref int RefReturn(ref int p)
{
    return ref p;
}
```

Again, this is equivalent to incrementing `x`, so the output is 11. In addition to modifying the resulting variable, you can use it as a `ref` argument to another method. To make our already purely demonstrative example even sillier, you could call `RefReturn` with the result of itself (twice):

```
RefReturn(ref RefReturn(ref RefReturn(ref x)))++;
```

`Ref` returns are valid for indexers as well as methods. This is most commonly useful to return an array element by reference, as shown in the following listing.

Listing 13.8 A ref return indexer exposing array elements

```
class ArrayHolder
{
    private readonly int[] array = new int[10];

    public ref int this[int index] => ref array[index];   ← Indexer returns an array
}                                         element by reference

static void Main()
{
    ArrayHolder holder = new ArrayHolder();
    ref int x = ref holder[0];
    ref int y = ref holder[0];                         ← Declares two ref locals referring
                                                       to the same array element

    x = 20;           ← Changes the array
    Console.WriteLine(y);   ← Observes the
}                                         value via x
```

You've now covered all the new features in C# 7.0, but later point releases expanded the set of `ref`-related features. The first feature was one I was quite frustrated by when writing my initial draft of this chapter: lack of conditional `? :` operator support.

13.2.3 The conditional `? :` operator and `ref` values (C# 7.2)

The conditional `? :` operator has been present since C# 1.0 and is familiar from other languages:

```
condition ? expression1 : expression2
```

It evaluates its first operand (the condition), and then evaluates either the second or third operand to provide the overall result. It feels natural to want to achieve the same thing with ref values, picking one or another variable based on a condition.

With C# 7.0, this wasn't feasible, but it is in C# 7.2. A conditional operator can use ref values for the second and third operands, at which point the result of the conditional operator is also a variable that can be used with the `ref` modifier. As an example, the following listing shows a method that counts the even and odd values in a sequence, returning the result as a tuple.

Listing 13.9 Counting even and odd elements in a sequence

```
static (int even, int odd) CountEvenAndOdd(IEnumerable<int> values)
{
    var result = (even: 0, odd: 0);
    foreach (var value in values)
    {
        ref int counter = ref (value & 1) == 0 ?           | Picks the appropriate
                                                    ref result.even : ref result.odd;      | variable to increment
        counter++;                                         | Increments it
    }
    return result;
}
```

The use of a tuple here is somewhat coincidental, although it serves to demonstrate how useful it is for tuples to be mutable. This addition makes the language feel much more consistent. The result of the conditional operator can be used as an argument to a ref parameter, assigned to a ref local, or used in a ref return. It all just drops out nicely. The next C# 7.2 feature addresses an issue you looked at in section 13.2.1 when discussing the restrictions on ref locals: how do you take a reference to a read-only variable?

13.2.4 Ref readonly (C# 7.2)

So far, all the variables you've been aliasing have been writable. In C# 7.0, that's all that's available. But that's insufficient in two parallel scenarios:

- You may want to alias a read-only field for the sake of efficiency to avoid copying.
- You may want to allow only read-only access via the ref variable.

The introduction of `ref readonly` in C# 7.2 addresses both scenarios. Both ref locals and ref returns can be declared with the `readonly` modifier, and the result is read-only, just like a read-only field. You can't assign a new value to the variable, and if it's a struct type, you can't modify any fields or call property setters.

TIP Given that one of the reasons for using `ref readonly` is to avoid copying, you could be surprised to hear that sometimes it has the opposite effect. You'll look at this in detail in section 13.4. Don't start using `ref readonly` in your production code without reading that section!

The two places you can put the modifier work together: if you call a method or indexer with a `ref readonly` return and want to store the result in a local variable, it has to be a `ref readonly` local variable, too. The following listing shows how the read-only aspects chain together.

Listing 13.10 `ref readonly` return and local

```
static readonly int field = DateTime.UtcNow.Second;           ← Initializes a read-only field
                                                               with an arbitrary value
static ref readonly int GetFieldAlias() => ref field;      ←
                                                               Returns a read-only
                                                               alias to the field
static void Main()
{
    ref readonly int local = ref GetFieldAlias();           ←
    Console.WriteLine(local);                               ← initializes a read-only ref
                                                               local using the method
}
```

This works with indexers, too, and it allows immutable collections to expose their data directly without any copying but without any risk of the memory being mutated. Note that you can return a `ref readonly` without the underlying variable being read-only, which provides a read-only view over an array, much like `ReadOnlyCollection` does for arbitrary collections but with copy-free read access. The following listing shows a simple implementation of this idea.

Listing 13.11 A read-only view over an array with copy-free reads

```
class ReadOnlyArrayView<T>
{
    private readonly T[] values;

    public ReadOnlyArrayView(T[] values) =>
        this.values = values;                                | Copies the array reference
                                                               without cloning contents

    public ref readonly T this[int index] =>               | Returns a read-only alias
        ref values[index];                                to the array element
    }

    ...
    static void Main()
    {
        var array = new int[] { 10, 20, 30 };
        var view = new ReadOnlyArrayView<int>(array);

        ref readonly int element = ref view[0];
        Console.WriteLine(element);
        array[0] = 100;
        Console.WriteLine(element);                          | Modification to the array
                                                               is visible via the local.
    }
}
```

This example isn't compelling in terms of efficiency gains because `int` is already a small type, but in scenarios using larger structs to avoid excessive heap allocation and garbage collection, the benefits can be significant.

Implementation details

In IL, a `ref readonly` method is implemented as a regular `ref`-returning method (the return type is a `by-ref` type) but with `[InAttribute]` from the `System.Runtime.InteropServices` namespace applied to it. This attribute is, in turn, specified with the `modreq` modifier in IL: if a compiler isn't aware of `InAttribute`, it should reject any call to the method. This is a safety mechanism to prevent misuse of the method's return value. Imagine a C# 7.0 compiler (one that's aware of `ref` returns but not `ref readonly` returns) trying to call a `ref readonly` returning method from another assembly. It could allow the caller to store the result in a writable `ref` local and then modify it, thereby violating the intention of the `ref readonly` return.

You can't declare `ref readonly` returning methods unless `InAttribute` is available to the compiler. That's rarely an issue, because it's been in the desktop framework since .NET 1.1 and in .NET Standard 1.1. If you absolutely have to, you can declare your own attribute in the right namespace, and the compiler will use that.

The `readonly` modifier can be applied to local variables and return types as you've seen, but what about parameters? If you have a `ref readonly` local and want to pass it into a method without just copying the value, what are your options? You might expect the answer to be the `readonly` modifier again, just applied to parameters, but reality is slightly different, as you'll see in the next section.

13.3 in parameters (C# 7.2)

C# 7.2 adds `in` as a new modifier for parameters in the same style as `ref` or `out` but with a different intention. When a parameter has the `in` modifier, the intention is that the method won't change the parameter value, so a variable can be passed by reference to avoid copying. Within the method, an `in` parameter acts like a `ref readonly` local variable. It's still an alias for a storage location passed by the caller, so it's important that the method doesn't modify the value; the caller would see that change, which goes against the point of it being an `in` parameter.

There's a big difference between an `in` parameter and a `ref` or `out` parameter: the caller doesn't have to specify the `in` modifier for the argument. If the `in` modifier is missing, the compiler will pass the argument by reference if the argument is a variable but take a copy of the value as a hidden local variable and pass that by reference if necessary. If the caller specifies the `in` modifier explicitly, the call is valid only if the argument can be passed by reference directly. The following listing shows all the possibilities.

Listing 13.12 Valid and invalid possibilities for passing arguments for `in` parameters

```
static void PrintDateTime(in DateTime value)
{
    string text = value.ToString(
        "yyyy-MM-dd'T'HH:mm:ss",
        CultureInfo.InvariantCulture);
```

← Declares method with `in` parameter

```

    Console.WriteLine(text);
}

static void Main()
{
    DateTime start = DateTime.UtcNow;
    PrintDateTime(start);
    PrintDateTime(in start);
    PrintDateTime(start.AddMinutes(1));
    PrintDateTime(in start.AddMinutes(1));
}

```

Variable is passed by reference implicitly.

Variable is passed by reference explicitly (due to `in` modifier).

Result is copied to hidden local variable, which is passed by reference.

Compile-time error: argument can't be passed by reference.

In the generated IL, the parameter is equivalent to a `ref` parameter decorated with `[IsReadOnlyAttribute]` from the `System.Runtime.CompilerServices` namespace. This attribute was introduced much more recently than `InAttribute`; it's in .NET 4.7.1, but it's not even in .NET Standard 2.0. It'd be annoying to have to either add a dependency or declare the attribute yourself, so the compiler generates the attribute in your assembly automatically if it's not otherwise available.

The attribute doesn't have the `modreq` modifier in IL; any C# compiler that doesn't understand `IsReadOnlyAttribute` will treat it as a regular `ref` parameter. (The CLR doesn't need to know about the attribute either.) Any callers recompiled with a later version of a compiler will suddenly fail to compile, because they'll now require the `in` modifier instead of the `ref` modifier. That leads us to a bigger topic of backward compatibility.

13.3.1 Compatibility considerations

The way that the `in` modifier is optional at the call site leads to an interesting backward-compatibility situation. Changing a method parameter from being a value parameter (the default, with no modifiers) to an `in` parameter is always *source* compatible (you should always be able to recompile without changing calling code) but is never *binary* compatible (any existing compiled assemblies calling the method will fail at execution time). Exactly what that means will depend on your situation. Suppose you want to change a method parameter to be an `in` parameter for an assembly that has already been released:

- If your method is accessible to callers outside your control (if you're publishing a library to NuGet, for example), this is a breaking change and should be treated like any other breaking change.
- If your code is accessible only to callers that will definitely be recompiled when they use the new version of your assembly (even if you can't change that calling code), then this won't break those callers.
- If your method is only internal to your assembly,¹ you don't need to worry about binary compatibility because all the callers will be recompiled anyway.

¹ If your assembly uses `InternalsVisibleTo`, the situation is more nuanced; that level of detail is beyond the scope of this book.

Another slightly less likely scenario exists: if you have a method with a `ref` parameter purely for the sake of avoiding copying (you never modify the parameter in the method), changing that to an `in` parameter is always *binary* compatible, but never *source* compatible. That's the exact opposite of changing a value parameter to an `in` parameter.

All of this assumes that the act of using an `in` parameter doesn't break the semantics of the method itself. That's not always a valid assumption; let's see why.

13.3.2 The surprising mutability of `in` parameters: External changes

So far, it sounds like if you don't modify a parameter within a method, it's safe to make it an `in` parameter. That's not the case, and it's a dangerous expectation. The compiler stops the *method* from modifying the parameter, but it can't do anything about other code modifying it. You need to remember that an `in` parameter is an alias for a storage location that other code may be able to modify. Let's look at a simple example first, which may seem utterly obvious.

Listing 13.13 `in` parameter and value parameter differences in the face of side effects

```
static void InParameter(in int p, Action action)
{
    Console.WriteLine("Start of InParameter method");
    Console.WriteLine($"p = {p}");
    action();
    Console.WriteLine($"p = {p}");
}

static void ValueParameter(int p, Action action)
{
    Console.WriteLine("Start of ValueParameter method");
    Console.WriteLine($"p = {p}");
    action();
    Console.WriteLine($"p = {p}");
}

static void Main()
{
    int x = 10;
    InParameter(x, () => x++);
    ValueParameter(x, () => x++);
}
```

The first two methods are identical except for the log message displayed and the nature of the parameter. In the `Main` method, you call the two methods in the same way, passing in a local variable with an initial value of 10 as the argument and an action that increments the variable. The output shows the difference in semantics:

```
Start of InParameter method
p = 10
p = 11
```

```
Start of ValueParameter method
p = 10
p = 10
```

As you can see, the `InParameter` method is able to observe the change caused by calling `action()`; the `ValueParameter` method isn't. This isn't surprising; `in` parameters are intended to share a storage location, whereas value parameters are intended to take a copy.

The problem is that although it's obvious in this particular case because there's so little code, in other examples it might not be. For example, the `in` parameter could happen to be an alias for a field in the same class. In that case, any modifications to the field, either directly in the method or by other code that the method calls, will be visible via the parameter. That isn't obvious either in the calling code or the method itself. It gets even harder to predict what will happen when multiple threads are involved.

I'm deliberately being somewhat alarmist here, but I think this is a real problem. We're used to highlighting the possibility of this sort of behavior² with `ref` parameters by specifying the modifier on the parameter and the argument. Additionally, the `ref` modifier feels like it's implicitly concerned with how changes in a parameter are visible, whereas the `in` modifier is about *not* changing the parameter. In section 13.3.4, I'll give more guidance on using `in` parameters, but for the moment you should just be aware of the potential risk of the parameter changing its value unexpectedly.

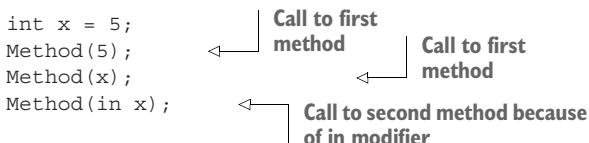
13.3.3 Overloading with `in` parameters

One aspect I haven't touched on yet is method overloading: what happens if you want two methods with the same name and the same parameter type, but in one case the parameter is an `in` parameter and in the second method it's not?

Remember that as far as the CLR is concerned, this is just another `ref` parameter. You can't overload the method by just changing between `ref`, `out`, and `in` modifiers; they all look the same to the CLR. But you can overload an `in` parameter with a regular value parameter:

```
void Method(int x) { ... }
void Method(in int x) { ... }
```

New tiebreaker rules in overload resolution make the method with the value parameter better with respect to an argument that doesn't have an `in` modifier:



² I like to think of it as being similar to the quantum entanglement phenomenon known as “spooky action at a distance.”

These rules allow you to add overloads for existing method names without too many compatibility concerns if the existing methods have value parameters and the new methods have in parameters.

13.3.4 Guidance for in parameters

Full disclosure: I haven't used in parameters in real code yet. The guidance here is speculative.

The first thing to note is that in parameters are intended to improve performance. As a general principle, I wouldn't start making any changes to your code to improve performance before you've *measured* performance in a meaningful and repeatable way and set goals for it. If you're not careful, you can complicate your code in the name of optimization, only to find out that even if you massively improved the performance of one or two methods, those methods weren't on a critical path for the application anyway. The exact goals you have will depend on the kind of code you're writing (games, web applications, libraries, IoT applications, or something else), but careful measurement is important. For microbenchmarks, I recommend the BenchmarkDotNet project.

The benefit of in parameters lies in reducing the amount of data that needs to be copied. If you're using only reference types or small structs, no improvement may occur at all; logically, the storage location still needs to be passed to the method, even if the value at that storage location isn't being copied. I won't make too many claims here because of the black box of JIT compilation and optimization. Reasoning about performance without testing it is a bad idea: enough complex factors are involved to turn that reasoning into an educated guess at best. I'd expect the benefits of in parameters to increase as the size of the structs involved increases, however.

My main concern about in parameters is that they can make reasoning about your code much harder. You can read the value of the same parameter twice and get different results, despite your method not changing anything, as you saw in section 13.3.2. That makes it harder to write correct code and easy to write code that appears to be correct but isn't.

There's a way to avoid this while still getting many of the benefits of in parameters, though: by carefully reducing or removing the possibilities of them changing. If you have a public API that's implemented via a deep stack of private method calls, you can use a value parameter for that public API and then use in parameters in the private methods. The following listing provides an example, although it's not doing any meaningful computations.

Listing 13.14 Using in parameters safely

```
public static double PublicMethod(  
    LargeStruct first,  
    LargeStruct second)  
{  
    double firstResult = PrivateMethod(in first);  
}
```

Public method using
value parameters

```

        double secondResult = PrivateMethod(in second);
        return firstResult + secondResult;
    }

private static double PrivateMethod(           | Private method using
    in LargeStruct input)                      | an in parameter
{
    double scale = GetScale(in input);
    return (input.X + input.Y + input.Z) * scale;
}

private static double GetScale(in LargeStruct input) =>   | Another method with
    input.Weight * input.Score;                                | an in parameter

```

With this approach, you can guard against unexpected change; because all the methods are private, you can inspect all the callers to make sure they won't be passing in values that could change while your method is executing. A single copy of each struct will be made when `PublicMethod` is called, but those copies are then aliased for use in the private methods, isolating your code from any changes the caller may be making in other threads or as side effects of the other methods. In some cases, you may *want* the parameter to be changeable, but in a way that you carefully document and control.

Applying the same logic to internal calls is also reasonable but requires more discipline because there's more code that can call the method. As a matter of personal preference, I've explicitly used the `in` modifier at the call site as well as in the parameter declaration to make it obvious what's going on when reading the code.

I've summed all of this up in a short list of recommendations:

- Use `in` parameters only when there's a measurable and significant performance benefit. This is most likely to be true when large structs are involved.
- Avoid using `in` parameters in public APIs unless your method can function correctly even if the parameter values change arbitrarily during the method.
- Consider using a public method as a barrier against change and then using `in` parameters within the private implementation to avoid copying.
- Consider explicitly using the `in` modifier when calling a method that takes an `in` parameter unless you're deliberately using the compiler's ability to pass a hidden local variable by reference.

Many of these guidelines could be easily checked by a Roslyn analyzer. Although I don't know of such an analyzer at the time of this writing, I wouldn't be surprised to see a NuGet package become available.

NOTE If you detect an implicit challenge here, you're right. If you let me know about an analyzer like this, I'll add a note on the website.

All of this depends on the amount of copying genuinely being reduced, and that's not as straightforward as it sounds. I alluded to this earlier, but now it's time to look much more closely at when the compiler implicitly copies structs and how you can avoid that.

13.4 Declaring structs as readonly (C# 7.2)

The point of `in` parameters is to improve performance by reducing copying for structs. That sounds great, but an obscure aspect of C# gets in our way unless we're careful. We'll look at the problem first and then at how C# 7.2 solves it.

13.4.1 Background: Implicit copying with read-only variables

C# has been implicitly copying structs for a long time. It's all documented in the specification, but I wasn't aware of it until I spotted a mysterious performance boost in Noda Time when I'd accidentally forgotten to make a field read-only.

Let's take a look at a simple example. You're going to declare a `YearMonthDay` struct with three read-only properties: `Year`, `Month`, and `Day`. You're not using the built-in `DateTime` type for reasons that will become clear later. The following listing shows the code for `YearMonthDay`; it's really simple. (There's no validation; it's purely for demonstration in this section.)

Listing 13.15 A trivial year/month/day struct

```
public struct YearMonthDay
{
    public int Year { get; }
    public int Month { get; }
    public int Day { get; }

    public YearMonthDay(int year, int month, int day) =>
        (Year, Month, Day) = (year, month, day);
}
```

Now let's create a class with two `YearMonthDay` fields: one read-only and one read-write. You'll then access the `Year` property in both fields.

Listing 13.16 Accessing properties via a read-only or read-write field

```
class ImplicitFieldCopy
{
    private readonly YearMonthDay readOnlyField =
        new YearMonthDay(2018, 3, 1);
    private YearMonthDay readWriteField =
        new YearMonthDay(2018, 3, 1);

    public void CheckYear()
    {
        int readOnlyFieldYear = readOnlyField.Year;
        int readWriteFieldYear = readWriteField.Year;
    }
}
```

The IL generated for the two property accesses is different in a subtle but important way. Here's the IL for the read-only field; I've removed the namespaces from the IL for simplicity:

```
ldfld valuetype YearMonthDay ImplicitFieldCopy::readOnlyField
stloc.0
ldloca.s V_0
call instance int32 YearMonthDay::get_Year()
```

It loads the value of the field, thereby copying it to the stack. Only then can it call the `get_Year()` member, which is the getter for the `Year` property. Compare that with the code using the read-write field:

```
ldflda valuetype YearMonthDay ImplicitFieldCopy::readWriteField
call instance int32 YearMonthDay::get_Year()
```

This uses the `ldflda` instruction to load the address of the field onto the stack rather than `ldfld`, which loads the value of the field. This is only IL, which isn't what your computer executes directly. It's entirely possible that in some cases the JIT compiler is able to optimize this away, but in Noda Time I found that making fields read-write (with an attribute purely to explain why they weren't read-only) made a significant difference in performance.

The reason the compiler takes this copy is to avoid a read-only field being mutated by the code within the property (or method, if you're calling one). The intention of a read-only field is that nothing can change its value. It'd be odd if `readOnlyField.SomeMethod()` was able to modify the field. C# is designed to expect that any property setters will mutate the data, so they're prohibited entirely for read-only fields. But even a property getter could try to mutate the value. Taking a copy is a safety measure, effectively.

This affects only value types

Just as a reminder, it's fine to have a read-only field that's a reference type and for methods to mutate the data in the objects they refer to. For example, you could have a read-only `StringBuilder` field, and you'd still be able to append to that `StringBuilder`. The value of the field is only the reference, and that's what can't change.

In this section, we're focusing on the field type being a value type like `decimal` or `DateTime`. It doesn't matter whether the type that contains the field is a class or a struct.

Until C# 7.2, only fields could be read-only. Now we have `ref readonly` local variables and `in` parameters to worry about. Let's write a method that prints out the year, month, and day from a value parameter:

```
private void PrintYearMonthDay(YearMonthDay input) =>
    Console.WriteLine($"{input.Year} {input.Month} {input.Day}");
```

The IL for this uses the address of the value that's already on the stack. Each property access looks as simple as this:

```
ldarga.s input
call instance int32 Chapter13.YearMonthDay::get_Year()
```

This doesn't create any additional copies. The assumption is that if the property mutates the value, it's okay for your `input` variable to be changed; it's just a read-write variable, after all. But if you decide to change `input` to an `in` parameter like this, things change:

```
private void PrintYearMonthDay(in YearMonthDay input) =>
    Console.WriteLine($"{input.Year} {input.Month} {input.Day}");
```

Now in the IL for the method, each property access has code like this:

```
ldarg.1
ldobj Chapter13.YearMonthDay
stloc.0
ldloca.s V_0
call instance int32 YearMonthDay::get_Year()
```

The `ldobj` instruction copies the value from the address (the parameter) onto the stack. You were trying to avoid one copy being made by the caller, but in doing so you've introduced three copies within the method. You'd see the exact same behavior with `readonly ref` local variables, too. That's not good! As you've probably guessed, C# 7.2 has a solution to this: read-only structs to the rescue!

13.4.2 The `readonly` modifier for structs

To recap, the reason the C# compiler needs to make copies for read-only value type variables is to avoid code within those types changing the value of the variable. What if the struct could promise that it didn't do that? After all, most structs are designed to be immutable. In C# 7.2, you can apply the `readonly` modifier to a struct declaration to do exactly that.

Let's modify our year/month/day struct to be read-only. It's already obeying the semantics within the implementation, so you just need to add the `readonly` modifier:

```
public readonly struct YearMonthDay
{
    public int Year { get; }
    public int Month { get; }
    public int Day { get; }

    public YearMonthDay(int year, int month, int day) =>
        (Year, Month, Day) = (year, month, day);
}
```

After that simple change to the declaration, and without any changes to the code using the struct, the IL generated for `PrintYearMonthDay(in YearMonthDay input)` becomes more efficient. Each property access now looks like this:

```
ldarg.1
call instance int32 YearMonthDay::get_Year()
```

Finally, you've managed to avoid copying the whole struct even once.

If you look in the downloadable source code that accompanies the book, you'll see this in a separate struct declaration: `ReadOnlyYearMonthDay`. That was necessary so I could have samples with before and after, but in your own code you can just make an existing struct read-only without breaking source or binary compatibility. Going in the opposite direction is an insidious breaking change, however; if you decide to remove the modifier and modify an existing member to mutate the state of the value, code that was previously compiled expecting the struct to be read-only could end up mutating read-only variables in an alarming way.

You can apply the modifier only if your struct is genuinely read-only and therefore meets the following conditions:

- Every instance field and automatically implemented instance property must be read-only. Static fields and properties can still be read-write.
- You can assign to `this` only within constructors. In specification terms, `this` is treated as an `out` parameter in constructors, a `ref` parameter in members of regular structs, and an `in` parameter in members of read-only structs.

Assuming you already intended your structs to be read-only, adding the `readonly` modifier allows the compiler to help you by checking that you aren't violating that. I'd expect most user-defined structs to work right away. Unfortunately, there's a slight wrinkle when it comes to Noda Time, which may affect you, too.

13.4.3 XML serialization is implicitly read-write

Currently, most of the structs in Noda Time implement `IXmlSerializable`. Unfortunately, XML serialization is defined in a way that's actively hostile to writing read-only structs. My implementation in Noda Time typically looks like this:

```
void IXmlSerializable.ReadXml(XmlReader reader)
{
    var pattern = /* some suitable text parsing pattern for the type */;
    var text = /* extract text from the XmlReader */;
    this = pattern.Parse(text).Value;
}
```

Can you see the problem? It assigns to `this` in the last line. That prevents me from declaring these structs with the `readonly` modifier, which saddens me. I have three options at the moment:

- Leave the structs as they are, which means `in` parameters and `ref readonly` locals are inefficient.
- Remove XML serialization from the next major version of Noda Time.
- Use unsafe code in `ReadXml` to violate the `readonly` modifier. The `System.Runtime.CompilerServices.Unsafe` package makes this simpler.

None of these options is pleasant, and there's no twist as I reveal a cunning way of satisfying all the concerns. At the moment, I believe that structs implementing `IXmlSerializable` can't be genuinely read-only. No doubt there are other interfaces that

are implicitly mutable in the same way that you might want to implement in a struct, but I suspect that `IXmlSerializable` will be the most common one.

The good news is that most readers probably aren't facing this issue. Where you can make your user-defined structs read-only, I encourage you to do so. Just bear in mind that it's a one-way change for public code; you can safely remove the modifier later only if you're in the privileged position of being able to recompile all the code that uses the struct. Our next feature is effectively tidying up consistency: providing the same functionality to extension methods that's already present in struct instance methods.

13.5 Extension methods with ref or in parameters (C# 7.2)

Prior to C# 7.2, the first parameter in any extension method had to be a value parameter. This restriction is partially lifted in C# 7.2 to embrace the new ref-like semantics more thoroughly.

13.5.1 Using ref/in parameters in extension methods to avoid copying

Suppose you have a large struct that you'd like to avoid copying around and a method that computes a result based on the values of properties in that struct—the magnitude of a 3D vector, for example. If the struct provides the method (or property) itself, you're fine, particularly if the struct is declared with the `readonly` modifier. You can avoid copying with no problems. But maybe you're doing something more complex that the authors of the struct hadn't considered. The samples in this section use a trivial read-only `Vector3D` struct introduced in the following listing. The struct just exposes X, Y, and Z properties.

Listing 13.17 A trivial Vector3D struct

```
public readonly struct Vector3D
{
    public double X { get; }
    public double Y { get; }
    public double Z { get; }

    public Vector3D(double x, double y, double z)
    {
        X = x;
        Y = y;
        Z = z;
    }
}
```

If you write your own method accepting the struct with an `in` parameter, you're fine. You can avoid copying, but it may be slightly awkward to call. For example, you might end up having to write something like this:

```
double magnitude = VectorUtilities.Magnitude(vector);
```

That would be ugly. You have extension methods, but a regular extension method like this would copy the vector on each call:

```
public static double Magnitude(this Vector3D vector)
```

It's unpleasant to have to choose between performance and readability. C# 7.2 comes to the rescue in a reasonably predictable way: you can write extension methods with a `ref` or `in` modifier on the first parameter. The modifier can appear before or after the `this` modifier. If you're only computing a value, you should use an `in` parameter, but you can also use `ref` if you want to be able to modify the value in the original storage location without having to create a new value and copy it in. The following listing provides two sample extension methods on a `Vector3D`.

Listing 13.18 Extension methods using `ref` and `in`

```
public static double Magnitude(this in Vector3D vec) =>
    Math.Sqrt(vec.X * vec.X + vec.Y * vec.Y + vec.Z * vec.Z);

public static void OffsetBy(this ref Vector3D orig, in Vector3D off) =>
    orig = new Vector3D(orig.X + off.X, orig.Y + off.Y, orig.Z + off.Z);
```

The parameter names are abbreviated more than I'm normally comfortable with to avoid long-winded formatting in the book. Note that the second parameter in the `OffsetBy` method is an `in` parameter; you're trying to avoid copying to as great an extent as you can.

It's simple to use the extension methods. The only possibly surprising aspect is that unlike regular `ref` parameters, there's no sign of the `ref` modifier when calling `ref` extension methods. The following listing uses both of the extension methods I've shown to create two vectors, offset the first vector by the second vector, and then display the resulting vector and its magnitude.

Listing 13.19 Calling `ref` and `in` extension methods

```
var vector = new Vector3D(1.5, 2.0, 3.0);
var offset = new Vector3D(5.0, 2.5, -1.0);

vector.OffsetBy(offset);

Console.WriteLine($"({vector.X}, {vector.Y}, {vector.Z})");
Console.WriteLine(vector.Magnitude());
```

The output is as follows:

```
(6.5, 4.5, 2)
8.15475321515004
```

This shows that the call to `OffsetBy` modified the `vector` variable as you intended it to.

NOTE The `OffsetBy` method makes our immutable `Vector3D` struct feel somewhat mutable. This feature is still in its early days, but I suspect I'll feel much more comfortable writing extension methods with initial `in` parameters than with `ref` parameters.

An extension method with an initial `in` parameter can be called on a read-write variable (as you've seen by calling `vector.Magnitude()`), but an extension method with an initial `ref` parameter can't be called on a read-only variable. For example, if you create a read-only alias for `vector`, you can't call `OffsetBy`:

```
ref readonly var alias = ref vector;
alias.OffsetBy(offset);
```

Error: trying to use a
read-only variable as ref

Unlike regular extension methods, restrictions exist about the extended type (the type of the first parameter) for initial `ref` and `in` parameters.

13.5.2 Restrictions on ref and in extension methods

Normal extension methods can be declared to extend any type. They can use either a regular type or a type parameter with or without constraints:

```
static void Method(this string target)
static void Method(this IDisposable target)
static void Method<T>(this T target)
static void Method<T>(this T target) where T : IComparable<T>
static void Method<T>(this T target) where T : struct
```

In contrast, `ref` and `in` extension methods always have to extend value types. In the case of `in` extension methods, that value type can't be a type parameter either. These are valid:

```
static void Method(this ref int target)
static void Method<T>(this ref T target) where T : struct
static void Method<T>(this ref T target) where T : struct, IComparable<T>
static void Method<T>(this ref int target, T other)
static void Method(this in int target)
static void Method(this in Guid target)
static void Method<T>(this in Guid target, T other)
```

But these are invalid:

```
static void Method(this ref string target)
static void Method<T>(this ref T target)
    where T : IComparable<T>
static void Method<T>(this in string target)
static void Method<T>(this in T target)
    where T : struct
```

Note the difference between `in` and `ref`, where a `ref` parameter can be a type parameter so long as it has the `struct` constraint. An `in` extension method can still be generic (as per the final valid example), but the extended type can't be a type

parameter. At the moment, there's no constraint that can require that `T` is a `readonly struct`, which would be required for a generic `in` parameter to be useful. That may change in future versions of C#.

You may wonder why the extended type is constrained to be a value type at all. There are two primary reasons for this:

- The feature is designed to avoid expensive copying of value types, so there's no benefit for reference types.
- If a `ref` parameter could be a reference type, it could be set to a null reference within the method. That would disrupt an assumption C# developers and tooling can always make at the moment: that calling `x.Method()` (where `x` is a variable of some reference type) can never make `x` null.

I don't expect to use `ref` and `in` extension methods very much, but they do provide a pleasant consistency to the language.

The features in the remainder of the chapter are somewhat different from the ones you've examined so far. Just to recap, so far you've looked at these:

- Ref locals
- Ref returns
- Read-only versions of ref locals and ref returns
- `in` parameters: read-only versions of `ref` parameters
- Read-only structs, which allow `in` parameters and read-only ref locals and returns to avoid copying
- Extension methods targeting `ref` or `in` parameters

If you started with `ref` parameters and wondered how to extend the concept further, you might have come up with something similar to this list. We're now going to move on to `ref-like` structs, which are related to all of these but also feel like a whole new kind of type.

13.6 **Ref-like structs (C# 7.2)**

C# 7.2 introduces the notion of a `ref-like` struct: one that's intended to exist only on the stack. Just as with custom task types, it's likely that you'll never need to declare your own `ref-like` struct, but I expect C# code written against up-to-date frameworks in the next few years to use the ones built into the framework quite a lot.

First, you'll look at the basic rules for `ref-like` structs and then see how they're used and the framework support for them. I should note that these are a simplified form of the rules; consult the language specification for the gory details. I suspect that relatively few developers will need to know exactly how the compiler enforces the stack safety of `ref-like` structs, but it's important to understand the principle of what it's trying to achieve:

A `ref-like` struct value must stay on the stack, always.

Let's start by creating a ref-like struct. The declaration is the same as a normal struct declaration with the addition of the `ref` modifier:

```
public ref struct RefLikeStruct
{
    
    Struct members as normal
}
```

13.6.1 Rules for ref-like structs

Rather than say what you *can* do with it, here are some of the things you *can't* do with `RefLikeStruct` and a brief explanation:

- You can't include a `RefLikeStruct` as a field of any type that isn't also a ref-like struct. Even a regular struct can easily end up on the heap either via boxing or by being a field in a class. Even within another ref-like struct, you can use `RefLikeStruct` only as the type of an instance field—never a static field.
- You can't box a `RefLikeStruct`. Boxing is precisely designed to create an object on the heap, which is exactly what you don't want.
- You can't use `RefLikeStruct` as a type argument (either explicitly or by type inference) for any generic method or type, including as a type argument for a generic ref-like struct type. Generic code can use generic type arguments in all kinds of ways that put values on the heap, such as creating a `List<T>`.
- You can't use `RefLikeStruct[]` or any similar array type as the operand for the `typeof` operator.
- Local variables of type `RefLikeStruct` can't be used anywhere the compiler might need to capture them on the heap in a special generated type. That includes the following:
 - Async methods, although this could potentially be relaxed so a variable could be declared and used between await expressions, so long as it was never used across an await expression (with a declaration before the await and a usage after it). Parameters for async methods can't be ref-like struct types.
 - Iterator blocks, which already appear to have the “only using `RefLikeStruct` between two yield expressions is okay” rules. Parameters for iterator blocks can't be ref-like struct types.
 - Any local variable captured by a local method, LINQ query expression, anonymous method, or lambda expression.

Additionally, complicated rules³ indicate how ref local variables of ref-like types can be used. I suggest trusting the compiler here; if your code fails to compile because of ref-like structs, you're likely trying to make something available at a point where it will no

³ Translation: I'm finding them hard to understand. I understand the general purpose, but the complexity required to prevent bad things from happening is beyond my current level of interest in going over the rules line by line.

longer be alive on the stack. With this set of rules keeping values on the stack, you can finally look at using the poster child for ref-like structs: `Span<T>`.

13.6.2 `Span<T>` and `stackalloc`

There are several ways of accessing chunks of memory in .NET. Arrays are the most common, but `ArraySegment<T>` and pointers are also used. One large downside of using arrays directly is that the array effectively owns all its memory; an array is never just part of a larger piece of memory. That doesn't sound too bad until you think of how many method signatures you've seen like this:

```
int ReadData(byte[] buffer, int offset, int length)
```

This “buffer, offset, length” set of parameters occurs all over the place in .NET, and it's effectively a code smell suggesting that we haven't had the right abstraction in place. `Span<T>` and the related types aim to fix this.

NOTE Some uses of `Span<T>` will work just by adding a reference to the `System.Memory` NuGet package. Others require framework support. The code presented in this section has been built against .NET Core 2.1. Some listings will build against earlier versions of the framework as well.

`Span<T>` is a ref-like struct that provides read/write, indexed access to a section of memory just like an array but without any concept of owning that memory. A span is always created from something else (maybe a pointer, maybe an array, even data created directly on the stack). When you use a `Span<T>`, you don't need to care where the memory has been allocated. Spans can be *sliced*: you can create one span as a subsection of another without copying any data. In new versions of the framework, the JIT compiler will be aware of `Span<T>` and handle it in a heavily optimized manner.

The ref-like nature of `Span<T>` sounds irrelevant, but it has two significant benefits:

- It allows a span to refer to memory with a tightly controlled lifecycle, as the span can't escape from the stack. The code that allocates the memory can pass a span to other code and then free the memory afterward with confidence that there won't be any spans left to refer to that now-deallocated memory.
- It allows custom one-time initialization of data in a span without any copying and without the risk of code being able to change the data afterward.

Let's demonstrate both of these points in a simple way by writing a method to generate a random string. Although `Guid.NewGuid` often can be used for this purpose, sometimes you may want a more customized approach using a different set of characters and length. The following listing shows the traditional code you might have used in the past.

Listing 13.20 Generating a random string by using a `char []`

```
static string Generate(string alphabet, Random random, int length)
{
    char[] chars = new char[length];
```

```

    for (int i = 0; i < length; i++)
    {
        chars[i] = alphabet[random.Next(alphabet.Length)];
    }
    return new string(chars);
}

```

Here's an example of calling the method to generate a string of 10 lowercase letters:

```

string alphabet = "abcdefghijklmnopqrstuvwxyz";
Random random = new Random();
Console.WriteLine(Generate(alphabet, random, 10));

```

Listing 13.20 performs two heap allocations: one for the char array and one for the string. The data needs to be copied from one place to the other when constructing the string. You can improve this slightly if you know you'll always be generating reasonably small strings, and if you're in a position to use unsafe code. In that situation, you can use `stackalloc`, as shown in the following listing.

Listing 13.21 Generating a random string by using `stackalloc` and a pointer

```

unsafe static string Generate(string alphabet, Random random, int length)
{
    char* chars = stackalloc char[length];
    for (int i = 0; i < length; i++)
    {
        chars[i] = alphabet[random.Next(alphabet.Length)];
    }
    return new string(chars);
}

```

This performs only one heap allocation: the string. The temporary buffer is stack allocated, but you need to use the `unsafe` modifier because you're using a pointer. Unsafe code takes me out of my comfort zone; although I'm reasonably confident that this code is okay, I wouldn't want to do anything much more complicated with pointers. There's still the copy from the stack allocated buffer to the string, too.

The good news is that `Span<T>` also supports `stackalloc` without any need for the `unsafe` modifier, as shown in the following listing. You don't need the `unsafe` modifier because you're relying on the rules for ref-like structs to keep everything safe.

Listing 13.22 Generating a random string by using `stackalloc` and a `Span<char>`

```

static string Generate(string alphabet, Random random, int length)
{
    Span<char> chars = stackalloc char[length];
    for (int i = 0; i < length; i++)
    {
        chars[i] = alphabet[random.Next(alphabet.Length)];
    }
    return new string(chars);
}

```

That makes me more confident, but it's no more efficient; you're still copying data in a way that feels redundant. You can do better. All you need is this factory method in `System.String`:

```
public static string Create<TState>(
    int length, TState state, SpanAction<char, TState> action)
```

That uses `SpanAction<T, TArg>`, which is a new delegate with this signature:

```
delegate void SpanAction<T, in TArg>(Span<T> span, TArg arg);
```

These two signatures may look a little odd to start with, so let's unpack what the implementation of `Create` does. It takes the following steps:

- 1 Allocates a string with the requested length
- 2 Creates a span that refers to the memory inside the string
- 3 Calls the action delegate, passing in whatever state the method was given and the span
- 4 Returns the string

The first thing to note is that our delegate is able to write to the content of a string. That sounds like it defies everything you know about the immutability of strings, but the `Create` method is in control here. Yes, you can write whatever you like to the string, just as you can create a new string with whatever content you want. But by the time the string is returned, the content is effectively baked into the string. You can't try to cheat by holding onto the `Span<char>` that's passed to the delegate, because the compiler makes sure it doesn't escape the stack.

That still leaves the odd part about the state. Why do you need to pass in state that's then passed back to our delegate? It's easiest to show you an example; the following listing uses the `Create` method to implement our random string generator.

Listing 13.23 Generating a random string with `string.Create`

```
static string Generate(string alphabet, Random random, int length) =>
    string.Create(length, (alphabet, random), (span, state) =>
    {
        var alphabet2 = state.alphabet;
        var random2 = state.random;
        for (int i = 0; i < span.Length; i++)
        {
            span[i] = alphabet2[random2.Next(alphabet2.Length)];
        }
    });
}
```

At first, it looks like a lot of pointless repetition occurs. The second argument to `string.Create` is `(alphabet, random)`, which puts the `alphabet` and `random` parameters into a tuple to act as the state. You then unpack these values from the tuple again in the lambda expression:

```
var alphabet2 = state.alphabet;
var random2 = state.random;
```

Why not just capture the parameters in the lambda expression? Using `alphabet` and `random` within the lambda expression would compile and behave correctly, so why bother using the extra `state` parameter?

Remember the point of using spans: you're trying to reduce heap allocations as well as copying. When a lambda expression captures a parameter or local variable, it has to create an instance of a generated class so that the delegate has access to those variables. The lambda expression in listing 13.23 doesn't need to capture anything, so the compiler can generate a static method and cache a single delegate instance to use every time `Generate` is called. All the state is passed via the parameters to `string.Create`, and because C# 7 tuples are value types, there's no allocation for that state.

At this point, your simple string generation method is as good as it's going to get: it requires a single heap allocation and no extra data copying. Your code just writes straight into the string data.

This is just one example of the kind of thing that `Span<T>` makes possible. Related types exist; `ReadOnlySpan<T>`, `Memory<T>`, and `ReadOnlyMemory<T>` are the most important ones. A full deep-dive into them is beyond the scope of this book.

Importantly, our optimization of the `Generate` method didn't need to change its signature at all. It was a pure implementation change isolated from anything else, and that's what makes me excited. Although passing large structs by reference throughout your codebase would help avoid excessive copying, that's an invasive optimization. I far prefer optimizations that I can perform in a piecemeal, targeted fashion.

Just as `string` gains extra methods to make use of spans, so will many other types. We now take it for granted that any I/O-based operation will have an `async` option available in the framework, and I expect the same to be true for spans over time; whenever they'd be useful, they'll be available. I expect third-party libraries will offer overloads accepting spans, too.

STACKALLOC WITH INITIALIZERS (C# 7.3)

While we're on the subject of stack allocation, C# 7.3 adds one extra twist: initializers. Whereas with previous versions you could use `stackalloc` only with a size you wanted to allocate, with C# 7.3 you can specify the content of the allocated space as well. This is valid for both pointers and spans:

```
Span<int> span = stackalloc int[] { 1, 2, 3 };
int* pointer = stackalloc int[] { 4, 5, 6 };
```

I don't believe this has any significant efficiency gains over allocating and then manually populating the space, but it's certainly simpler code to read.

PATTERN-BASED FIXED STATEMENTS (C# 7.3)

As a reminder, the `fixed` statement is used to obtain a pointer to memory, temporarily preventing the garbage collector from moving that data. Before C# 7.3, this could

be used only with arrays, strings, and taking the address of a variable. C# 7.3 allows it to be used with any type that has an accessible method called `GetPinnableReference` that returns a reference to an unmanaged type. For example, if you have a method returning a `ref int`, you can use that in a fixed statement like this:

```
fixed (int* ptr = value)           ← Calls
{                                value.GetPinnableReference
}                                ← Code using
                                the pointer
```

This isn't something most developers would normally implement themselves, even within the small proportion of developers who use unsafe code on a regular basis. As you might expect, the types you're most likely to use this with are `Span<T>` and `ReadOnlySpan<T>`, allowing them to interoperate with code that already uses pointers.

13.6.3 IL representation of ref-like structs

Ref-like structs are decorated with an `[IsRefLikeAttribute]` attribute that is again from the `System.Runtime.CompilerServices` namespace. If you're targeting a version of the framework that doesn't have the attribute available, it'll be generated in your assembly.

Unlike `in` parameters, the compiler doesn't use the `modreq` modifier to require any tools consuming the type to be aware of it; instead, it also adds an `[ObsoleteAttribute]` to the type with a fixed message. Any compiler that understands `[IsRefLikeAttribute]` can ignore the `[ObsoleteAttribute]` if it has the right text. If the type author wants to make the type obsolete, they just use `[ObsoleteAttribute]` as normal, and the compiler will treat it as any other obsolete type.

Summary

- C# 7 adds support for pass-by-reference semantics in many areas of the language.
- C# 7.0 included only the first few features; use C# 7.3 for the full range.
- The primary aim of the ref-related features is for performance. If you're not writing performance-critical code, you may not need to use many of these features.
- Ref-like structs allow the introduction of new abstractions in the framework, starting with `Span<T>`. These abstractions aren't just for high-performance scenarios; they're likely to affect a large proportion of .NET developers over time.

Concise code in C# 7



This chapter covers

- Declaring methods within methods
- Simplifying calls by using `out` parameters
- Writing numeric literals more readably
- Using `throw` as an expression
- Using default literals

C# 7 comes with large features that change the way we approach code: tuples, deconstruction, and patterns. It comes with complex but effective features that are squarely aimed at high-performance scenarios. It also comes with a set of small features that just make life a little bit more pleasant. There's no single feature in this chapter that's earth-shattering; each makes a small difference, and the combination of all of them can lead to beautifully concise, clear code.

14.1 Local methods

If this weren't *C# in Depth*, this section would be short indeed; you can write methods within methods. There's more to it than that, of course, but let's start with a simple example. The following listing shows a simple local method within a regular

Main method. The local method prints and then increments a local variable declared within Main, demonstrating that variable capture works with local methods.

Listing 14.1 A simple local method that accesses a local variable

```
static void Main()
{
    int x = 10;           ← Declares local variable
    PrintAndIncrementX(); | used within method
    PrintAndIncrementX(); | Calls local method twice
    Console.WriteLine($"After calls, x = {x}");

    void PrintAndIncrementX()
    {
        Console.WriteLine($"x = {x}");
        x++;
    }
}
```

This looks a bit odd when you see it for the first time, but you soon get used to it. Local methods can appear anywhere you have a block of statements: methods, constructors, properties, indexers, event accessors, finalizers, and even within anonymous functions or nested within another local method.

A local method declaration is like a normal method declaration but with the following restrictions:

- It can't have any access modifiers (`public` and so on).
- It can't have the `extern`, `virtual`, `new`, `override`, `static`, or `abstract` modifiers.
- It can't have any attributes (such as `[MethodImpl]`) applied to it.
- It can't have the same name as another local method within the same parent; there's no way to overload local methods.

On the other hand, a local method acts like standard methods in other ways, such as the following:

- It can be `void` or return a value.
- It can have the `async` modifier.
- It can have the `unsafe` modifier.
- It can be implemented via an iterator block.
- It can have parameters, including optional ones.
- It can be generic.
- It can refer to any enclosing type parameters.
- It can be the target of a method group conversion to a delegate type.

As shown in the listing 14.1, it's fine to declare the method after it's used. Local methods can call themselves or other local methods that are in scope. Positioning can still

be important, though, largely in terms of how the local methods refer to *captured variables*: local variables declared in the enclosing code but used in the local method.

Indeed, much of the complexity around local methods, both in language rules and implementation, revolves around the ability for them to read and write captured variables. Let's start off by talking about the rules that the language imposes.

14.1.1 Variable access within local methods

You've already seen that local variables in the enclosing block can be read and written, but there's more nuance to it than that. There are a lot of small rules here, but you don't need to worry about learning them exhaustively. Mostly of the time you won't even notice them, and you can refer back to this section if the compiler complains about code that you expect to be valid.

A LOCAL METHOD CAN CAPTURE ONLY VARIABLES THAT ARE IN SCOPE

You can't refer to a local variable outside its scope, which is, broadly speaking, the block in which it's declared. For example, suppose you want your local method to use an iteration variable declared in a loop; the local method itself has to be declared in the loop, too. As a trivial example, this isn't valid:

```
static void Invalid()
{
    for (int i = 0; i < 10; i++)
    {
        PrintI();
    }
    void PrintI() => Console.WriteLine(i);      ← | Unable to access i;
}                                         | it's not in scope.
```

But with the local method inside the loop, it's valid¹:

```
static void Valid()
{
    for (int i = 0; i < 10; i++)
    {
        PrintI();
        void PrintI() => Console.WriteLine(i);      ← | Local method declared
    }                                         | within loop; i is in scope.
}
```

A LOCAL METHOD MUST BE DECLARED AFTER THE DECLARATION OF ANY VARIABLES IT CAPTURES

Just as you can't use a variable earlier than its declaration in regular code, you can't use a captured variable in a local method until after its declaration, either. This rule is more for consistency than out of necessity; it would've been feasible to specify the

¹ It may be a little strange to read, but it's valid.

language to require that any calls to the method occur after the variable's declaration, for example, but it's simpler to require all access to occur after declaration. Here's another trivial example of invalid code:

```
static void Invalid()
{
    void PrintI() => Console.WriteLine(i);      ← CS0841: Can't use local
    int i = 10;                                variable i before it's declared
    PrintI();
}
```

Just moving the local method declaration to after the variable declaration (whether before or after the `PrintI()` call) fixes the error.

A LOCAL METHOD CAN'T CAPTURE REF PARAMETERS OF THE ENCLOSING METHOD

Just like anonymous functions, local methods aren't permitted to use reference parameters of their enclosing method. For example, this is invalid:

```
static void Invalid(ref int p)
{
    PrintAndIncrementP();
    void PrintAndIncrementP() =>
        Console.WriteLine(p++);      ← Invalid access to
}                                         reference parameter
```

The reason for this prohibition for anonymous functions is that the created delegate might outlive the variable being captured. In most cases, this reason wouldn't apply to local methods, but as you'll see later, it's possible for local methods to have the same kind of issue. In most cases, you can work around this by declaring an extra parameter in the local method and passing the reference parameter by reference again:

```
static void Valid(ref int p)
{
    PrintAndIncrement(ref p);
    void PrintAndIncrement(ref int x) => Console.WriteLine(x++);
}
```

If you don't need to modify the parameter within the local method, you can make it a value parameter instead.

As a corollary of this restriction (again, mirroring a restriction for anonymous functions), local methods declared within structs can't access `this`. Imagine that `this` is an implicit extra parameter at the start of every instance method's parameter list. For class methods, it's a value parameter; for struct methods, it's a reference parameter. Therefore, you can capture `this` in local methods in classes but not in structs. The same workaround applies as for other reference parameters.

NOTE I've provided an example in the source code accompanying the book in `LocalMethodUsingThisInStruct.cs`.

LOCAL METHODS INTERACT WITH DEFINITE ASSIGNMENT

The rules of definite assignment in C# are complicated, and local methods complicate them further. The simplest way to think about it is as if the method were inlined at any point where it's called. That impacts assignment in two ways.

First, if a method that reads a captured variable is called before it's definitely assigned, that causes a compile-time error. Here's an example that tries to print the value of a captured variable in two places: once before it's been assigned a value and once afterward:

```
static void AttemptToReadNotDefinitelyAssignedVariable()
{
    int i;
    void PrintI() => Console.WriteLine(i);           ← CS0165: Use of unassigned
    PrintI();                                         local variable 'i'
    i = 10;                                         ← No error: i is
    PrintI();                                         ← definitely assigned here.
}
```

Notice that it's the location of the call to `PrintI` that causes the error here; the location of the method declaration itself is fine. If you move the assignment to `i` before any calls to `PrintI()`, that's fine, even if it's still after the declaration of `PrintI()`.

Second, if a local method writes to a captured variable in all possible execution flows, the variable will be definitely assigned at the end of any call to that method. Here's an example that assigns a value within a local method but then reads it within the containing method:

```
static void DefinitelyAssignInMethod()
{
    int i;                                     Call to the method makes
    AssignI();                                ← i definitely assigned.
    Console.WriteLine(i);                      ← So it's fine to
    void AssignI() => i = 10;                  ← Method performs
                                                ← the assignment.
}
```

There are a couple of final points to make about local methods and variables, but this time the variables under discussion are not captured variables but fields.

LOCAL METHODS CAN'T ASSIGN READ-ONLY FIELDS

Read-only fields can be assigned values only in field initializers or constructors. That rule doesn't change with local methods, but it's made a little stricter: even if a local method is declared within a constructor, it doesn't count as being inside the constructor in terms of field initialization. This code is invalid:

```
class Demo
{
    private readonly int value;

    public Demo()
    {
        AssignValue();
    }
}
```

```
void AssignValue()
{
    value = 10;           ← Invalid assignment
}                         to read-only field
}
```

This restriction isn't likely to be a significant problem, but it's worth being aware of. It stems from the fact that the CLR hasn't had to change in order to support local methods. They're just a compiler transformation. This leads us to considering exactly how the compiler *does* implement local methods, particularly with respect to captured variables.

14.1.2 Local method implementations

Local methods don't exist at the CLR level.² The C# compiler converts local methods into regular methods by performing whatever transformations are required to make the final code behave according to the language rules. This section provides examples of the transformations implemented by Roslyn (the Microsoft C# compiler) and focuses on how captured variables are treated, as that's the most complex aspect of the transformation.

Implementation details: Nothing guaranteed here

This section really is about how the C# 7.0 version of Roslyn implements local methods. This implementation could change in future versions of Roslyn, and other C# compilers may use a different implementation. It also means there's quite a lot of detail here that you may not be interested in.

The implementation does have performance implications that may affect how comfortable you are with using local methods in performance-sensitive code. But as with all performance matters, you should be basing your decisions more on careful measurement than on theory.

Local methods feel like anonymous functions in the way they can capture local variables from their surrounding code. But significant differences in the implementation can make local methods rather more efficient in many cases. At the root of this difference is the lifetime of the local variables involved. If an anonymous function is converted into a delegate instance, that delegate could be invoked long after the method has returned, so the compiler has to perform tricks, hoisting the captured variables into a class and making the delegate refer to a method in that class.

Compare that with local methods: in most cases, the local method can be invoked only during the call of the enclosing method; you don't need to worry about it referring

² If a C# compiler were to target an environment where local methods did exist, all of the information in this section would probably be irrelevant for that compiler.

to captured variables after that call has completed. That allows for a more efficient, stack-based implementation with no heap allocations. Let's start reasonably simply with a local method that increments a captured variable by an amount specified as an argument to the local method.

Listing 14.2 Local method modifying a local variable

```
static void Main()
{
    int i = 0;
    AddToI(5);
    AddToI(10);
    Console.WriteLine(i);
    void AddToI(int amount) => i += amount;
}
```

What does Roslyn do with this method? It creates a private mutable struct with public fields to represent all the local variables in the same scope that are captured by any local method. In this case, that's just the `i` variable. It creates a local variable within the `Main` method of that struct type and passes the variable by reference to the regular method created from `AddToI` along with the declared `amount` parameter, of course. You end up with something like the following listing.

Listing 14.3 What Roslyn does with listing 14.2

```
private struct MainLocals ← Generated mutable struct to store
{
    public int i;                                the local variables from Main
}

static void Main()
{
    MainLocals locals = new MainLocals();          | Creates and uses a value of
    locals.i = 0;                                 | the struct within the method
    AddToI(5, ref locals);           | Passes the struct by reference
    AddToI(10, ref locals);          | to the generated method
    Console.WriteLine(locals.i);
}

static void AddToI(int amount, ref MainLocals locals) | Generated method
{                                                 to represent the
    locals.i += amount;                         original local method
}
```

As usual, the compiler generates unspeakable names for the method and the struct. Note that in this example, the generated method is static. That's the case when either the local method is originally contained in a static member or when it's contained in an instance member but the local method doesn't capture `this` (explicitly or implicitly by using instance members within the local method).

The important point about generating this struct is that the transformation is almost free in terms of performance: all the local variables that would've been on the stack before are still on the stack; they are just bunched together in a struct so that they can be passed by reference to the generated method. Passing the struct by reference has two benefits:

- It allows the local method to modify the local variables.
- However many local variables are captured, calling the local method is cheap. (Compare that with passing them all by value, which would mean creating a second copy of each captured local variable.)

All of this without any garbage being generated on the heap. Hooray! Now let's make things a little more complex.

CAPTURING VARIABLES IN MULTIPLE SCOPES

In an anonymous function, if local variables are captured from multiple scopes, multiple classes are generated with a field in each class representing the inner scope holding a reference to an instance of the class representing the outer scope. That wouldn't work with the struct approach for local methods that you just saw because of the copying involved. Instead, the compiler generates one struct for each scope containing a captured variable and uses a separate parameter for each scope. The following listing deliberately creates two scopes, so we can see how the compiler handles it.

Listing 14.4 Capturing variables from multiple scopes

```
static void Main()
{
    DateTime now = DateTime.UtcNow;
    int hour = now.Hour;
    if (hour > 5)
    {
        int minute = now.Minute;
        PrintValues();

        void PrintValues() =>
            Console.WriteLine($"hour = {hour}; minute = {minute}");
    }
}
```

I used a simple `if` statement to introduce a new scope rather than a `for` or `foreach` loop, because this made the translation simpler to represent reasonably accurately. The following listing shows the compiler how the compiler translates the local methods into regular ones.

Listing 14.5 What Roslyn does with listing 14.4

```
struct OuterScope
{
    public int hour;
}
```

**Generated struct
for outer scope**

```

struct InnerScope
{
    public int minute;
}

static void Main()
{
    DateTime now = DateTime.UtcNow; ←
    OuterScope outer = new OuterScope();
    outer.hour = now.Hour;
    if (outer.hour > 5)
    {
        InnerScope inner = new InnerScope();
        inner.minute = now.Minute;
        PrintValues(ref outer, ref inner);
    }
}

static void PrintValues(
    ref OuterScope outer, ref InnerScope inner) ←
{
    Console.WriteLine($"hour = {outer.hour}; minute = {inner.minute}");
}

```

Generated struct for inner scope

Uncaptured local variable

Creates and uses struct for outer scope variable hour

Creates and uses struct for inner scope variable minute

Passes both structs by reference to generated method

Generated method to represent the original local method

In addition to demonstrating how multiple scopes are handled, this listing shows that uncaptured local variables aren't included in the generated structs.

So far, we've looked at cases where the local method can execute only while the containing method is executing, which makes it safe for the local variables to be captured in this efficient way. In my experience, this covers most of the cases where I've wanted to use local methods. There are occasional exceptions to that safe situation, though.

PRISON BREAK! HOW LOCAL METHODS CAN ESCAPE THEIR CONTAINING CODE

Local methods behave like regular methods in four ways that can stop the compiler from performing the “keep everything on the stack” optimization we've discussed so far:

- They can be asynchronous, so a call that returns a task almost immediately won't necessarily have finished executing the logical operation.
- They can be implemented with iterators, so a call that creates a sequence will need to continue executing the method when the next value in the sequence is requested.
- They can be called from anonymous functions, which could in turn be called (as delegates) long after the original method has finished.
- They can be the targets of method group conversions, again creating delegates that can outlive the original method call.

The following listing shows a simple example of the last bullet point. A local `Count` method captures a local variable in its enclosing `CreateCounter` method. The `Count` method is used to create an `Action` delegate, which is then invoked after the `CreateCounter` method has returned.

Listing 14.6 Method group conversion of a local method

```

static void Main()
{
    Action counter = CreateCounter();
    counter();
    counter();                                | Invokes the delegate after
                                                | CreateCounter has finished
}

static Action CreateCounter()
{
    int count = 0;                            ← Local variable
    return Count;                            ← captured by Count
    void Count() => Console.WriteLine(count++); ← Method group conversion of
                                                | Count to an Action delegate
}

```

← Local method

You can't use a struct on the stack for `count` anymore. The stack for `CreateCounter` won't exist by the time the delegate is invoked. But this feels very much like an anonymous function now; you could've implemented `CreateCounter` by using a lambda expression instead:

```

static Action CreateCounter()
{
    int count = 0;
    return () => Console.WriteLine(count++);   ← Alternative implementation
                                                | using a lambda expression
}

```

That gives you a clue as to how the compiler can implement the local method: it can apply a similar transformation for the local method as it would for the lambda expression, as shown in the following listing.

Listing 14.7 What Roslyn does with listing 14.6

```

static void Main()
{
    Action counter = CreateCounter();
    counter();
    counter();
}

static Action CreateCounter()
{
    CountHolder holder = new CountHolder();      | Creates and initializes object
    holder.count = 0;                            | holding captured variables
    return holder.Count;                         ← Method group conversion of
                                                | instance method from holder
}

private class CountHolder                      ←
{
    public int count;                          ← Captured variable
    public void Count() => Console.WriteLine(count++); ← Private class with
                                                | captured variables and local method
}

```

← Local method is now an instance method in generated class

The same kind of transformation is performed if the local method is used within an anonymous function if it's an `async` method or if it's an iterator (with `yield` statements). The performance-minded may wish to be aware that `async` methods and iterators can end up generating multiple objects; if you're working hard to prevent allocations and you're using local methods, you may wish to pass parameters explicitly to those local methods instead of capturing local variables. An example of this is shown in the next section.

Of course, the set of possible scenarios is pretty huge; one local method could use a method conversion for another local method, or you could use a local method within an `async` method, and so on. I'm certainly not going to try to cover every possible case here. This section is intended to give you a good idea of the two kinds of transformation the compiler can use when dealing with captured variables. To see what it's doing with *your* code, use a decompiler or `ildasm`, remembering to disable any "optimizations" the decompiler might do for you. (Otherwise, it could easily just show the local method, which doesn't help you at all.) Now that you've seen what you can do with local methods and how the compiler handles them, let's consider when it's appropriate to use them.

14.1.3 Usage guidelines

There are two primary patterns to spot where local methods might be applicable:

- You have the same logic repeated multiple times in a method.
- You have a private method that's used from only one other method.

The second case is a special case of the first in which you've taken the time to refactor the common code already. But the first case can occur when there's enough local state to make that refactoring ugly. Local methods can make the extraction significantly more appealing because of the ability to capture local variables.

When refactoring an existing method to become a local method, I advise consciously taking a two-stage approach. First, move the single-use method into the code that uses it without changing its signature.³ Second, look at the method parameters: are all the calls to the method using the same local variables as arguments? If so, those are good candidates for using captured variables instead, removing the parameter from the local method. Sometimes you may even be able to remove the parameters entirely.

Depending on the number and size of the parameters, this second step could even have a performance impact. If you were previously passing large value types by value, those were being copied on each call. Using captured variables instead can eliminate that copy, which could be significant if the method is being called a lot.

The important point about local methods is that it becomes clear that they're an implementation detail of a method rather than of a type. If you have a private

³ Sometimes this requires changes to the type parameters in the signature. Often if you have one generic method calling another, when you move the second method into the first, it can just use the type parameters of the first. Listing 14.9 demonstrates this.

method that makes sense as an operation in its own right but happens to be used in only one place at the moment, you may be better off leaving it where it is. The payoff—in terms of logical type structure—is much bigger when a private method is tightly bound to a single operation and you can’t easily imagine any other circumstances where you’d use it.

ITERATOR/ASYNC ARGUMENT VALIDATION AND LOCAL METHOD OPTIMIZATION

One common example of this is when you have iterator or async methods and want to eagerly perform argument validation. For example, the Listing 14.8 provides a sample implementation of one overload of `Select` in LINQ to Objects. The argument validation isn’t in an iterator block, so it’s performed as soon as the method is called, whereas the `foreach` loop doesn’t execute at all until the caller starts iterating over the returned sequence.

Listing 14.8 Implementing Select without local methods

```
public static IEnumerable<TResult> Select<TSource, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, TResult> selector)
{
    Preconditions.CheckNotNull(source, nameof(source));
    Preconditions.CheckNotNull(
        selector, nameof(selector));
    return SelectImpl(source, selector);           ← Eagerly checks
}                                                 arguments
                                                Delegates to the
                                                implementation

private static IEnumerable<TResult> SelectImpl<TSource, TResult>(
    IEnumerable<TSource> source,
    Func<TSource, TResult> selector)
{
    foreach (TSource item in source)
    {
        yield return selector(item);           Implementation
    }                                         executes lazily
}
}
```

Now, with local methods available, you can move the implementation into the `Select` method, as shown in the following listing.

Listing 14.9 Implementing Select with a local method

```
public static IEnumerable<TResult> Select<TSource, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, TResult> selector)
{
    Preconditions.CheckNotNull(source, nameof(source));
    Preconditions.CheckNotNull(selector, nameof(selector));
    return SelectImpl(source, selector);

    IEnumerable<TResult> SelectImpl(
        IEnumerable<TSource> validatedSource,
```

```
Func<TSource, TResult> validatedSelector)
{
    foreach (TSource item in validatedSource)
    {
        yield return validatedSelector(item);
    }
}
```

I've highlighted one interesting aspect of the implementation: you still pass the (now-validated) parameters into the local method. This isn't required; you could make the local method parameterless and just use the captured source and selector variables, but it's a performance tweak—it reduces the number of allocations required. Is this performance difference important? Would the version using variable capture be significantly more readable? Answers to both questions depend on the context and are likely to be somewhat subjective.

READABILITY SUGGESTIONS

Local methods are still new enough to me that I'm slightly wary of them. I'm erring on the side of leaving code as it is rather than refactoring toward local methods at the moment. In particular, I'm avoiding using the following two features:

- Even though you can declare a local method within the scope of a loop or other block, I find that odd to read. I prefer to use local methods only when I can declare them right at the bottom of the enclosing method. I can't capture any variables declared within loops, but I'm okay with that.
- You can declare local methods within other local methods, but that feels like a rabbit hole I'd rather not go down.

Your tastes may vary, of course, but as always, I caution against using a new feature just because you can. (Experiment with it for the sake of experimentation, certainly, but don't let the new shiny things lure you into sacrificing readability.)

Time for some good news: the first feature of this chapter was the big one. The remaining features are much simpler.

14.2 Out variables

Before C# 7, `out` parameters were slightly painful to work with. An `out` parameter required a variable to already be declared before you could use it as an argument for the parameter. Because declarations are separate statements, this meant that in some places where you wanted a single expression—initializing a variable, for example—you had to reorganize your code to have multiple statements.

14.2.1 Inline variable declarations for out parameters

C# 7 removes this pain point by allowing new variables to be declared within the method invocation itself. As a trivial example, consider a method that takes textual input, attempts to parse it as an integer using `int.TryParse`, and then returns either

the parsed value as a nullable integer (if it parsed successfully) or null (if it didn't). In C# 6, this would have to be implemented using at least two statements: one to declare the variable and a second to call `int.TryParse` passing the newly declared variable for the `out` parameter:

```
static int? ParseInt32(string text)
{
    int value;
    return int.TryParse(text, out value) ? value : (int?) null;
}
```

In C# 7, the `value` variable can be declared within the method call itself, which means you can implement the method with an expression body:

```
static int? ParseInt32(string text) =>
    int.TryParse(text, out int value) ? value : (int?) null;
```

In several ways, `out` variable arguments behave similarly to variables introduced by pattern matches:

- If you don't care about the value, you can use a single underscore as the name to create a discard.
- You can use `var` to declare an implicitly typed variable (the type is inferred from the type of the parameter).
- You can't use an `out` variable argument in an expression tree.
- The scope of the variable is the surrounding block.
- You can't use `out` variables in field, property, or constructor initializers or in query expressions before C# 7.3. You'll look at an example of this shortly.
- The variable will be definitely assigned if (and only if) the method is definitely invoked.

To demonstrate the last point, consider the following code, which tries to parse two strings and sum the results:

```
static int? ParseAndSum(string text1, string text2) =>
    int.TryParse(text1, out int value1) &&
    int.TryParse(text2, out int value2)
    ? value1 + value2 : (int?) null;
```

In the third operand of the conditional operator, `value1` is definitely assigned (so you could return that if you like), but `value2` isn't definitely assigned; if the first call to `int.TryParse` returned `false`, you wouldn't call `int.TryParse` the second time because of the short-circuiting nature of the `&&` operator.

14.2.2 Restrictions lifted in C# 7.3 for out variables and pattern variables

As I mentioned in section 12.5, pattern variables can't be used when initializing fields or properties, in construction initializers (`this(...)` and `base(...)`), or in query expressions. The same restriction applies to `out` variables until C# 7.3, which lifts all

those restrictions. The following listing demonstrates this and shows that the result of the `out` variable is also available within the constructor body.

Listing 14.10 Using an `out` variable in a constructor initializer

```
class ParsedText
{
    public string Text { get; }
    public bool Valid { get; }

    protected ParsedText(string text, bool valid)
    {
        Text = text;
        Valid = valid;
    }
}

class ParsedInt32 : ParsedText
{
    public int? Value { get; }

    public ParsedInt32(string text)
        : base(text, int.TryParse(text, out int parseResult))
    {
        Value = Valid ? parseResult : (int?) null;
    }
}
```

Although the restrictions prior to C# 7.3 never bothered me, it's nice that they've now been removed. In the rare cases that you needed to use patterns or `out` variables for initializers, the alternatives were relatively annoying and usually involved creating a new method just for this purpose.

That's about it for `out` variable arguments. They're just a useful little shorthand to avoid otherwise-annoying variable declaration statements.

14.3 Improvements to numeric literals

Literals haven't changed much in the course of C#'s history. No changes at all occurred from C# 1 until C# 6, when interpolated string literals were introduced, but that didn't change numbers at all. In C# 7, two features are aimed at number literals, both for the sake of improving readability: binary integer literals and underscore separators.

14.3.1 Binary integer literals

Unlike floating-point literals (for `float`, `double`, and `decimal`), integer literals have always had two options for the base of the literal: you could use decimal (no prefix) or hex (a prefix of `0x` or `0X`).⁴ C# 7 extends this to binary literals, which use a prefix of

⁴ The C# designers wisely eschewed the ghastly octal literals that Java inherited from C. What's the value of `011?` Why, 9, "of course."

0b or 0B. This is particularly useful if you’re implementing a protocol with specific bit patterns for certain values. It doesn’t affect the execution-time behavior at all, but it can make the code a lot easier to read. For example, which of these three lines initializes a byte with the top bit and the bottom three bits set and the other bits unset?

```
byte b1 = 135;
byte b2 = 0x83;
byte b3 = 0b10000111;
```

They all do. But you can tell that easily in the third line, whereas the other two take slightly longer to check (at least for me). Even that last one takes longer than it might, because you still have to check that you have the right number of bits in total. If only there were a way of clarifying it even more.

14.3.2 Underscore separators

Let’s jump straight into underscore separators by improving the previous example. If you want to specify all the bits of a byte and do so in binary, it’s easier to spot that you have two nibbles than to count all eight bits. Here’s the same code with a fourth line that uses an underscore to separate the nibbles:

```
byte b1 = 135;
byte b2 = 0x83;
byte b3 = 0b10000111;
byte b4 = 0b1000_0111;
```

Love it! I can really check that at a glance. Underscore separators aren’t restricted to binary literals, though, or even to integer literals. You can use them in any numeric literal and put them (almost) anywhere within the literal. In decimal literals, you’re most likely to use them every three digits like thousands separators (at least in Western cultures). In hex literals, they’re generally most useful every two, four, or eight digits to separate 8-, 16-, or 32-bit parts within the literal. For example:

```
int maxInt32 = 2_147_483_647;
decimal largeSalary = 123_456_789.12m;
ulong alternatingBytes = 0xff_00_ff_00_ff_00_ff_00;
ulong alternatingWords = 0xffff_0000_ffff_0000;
ulong alternatingDwords = 0xffffffff_00000000;
```

This flexibility comes at a price: the compiler doesn’t check that you’re putting the underscores in sensible places. You can even put multiple underscores together. Valid but unfortunate examples include the following:

```
int wideFifteen = 1_____5;
ulong notQuiteAlternatingWords = 0xffff_000_ffff_0000;
```

You also should be aware of a few restrictions:

- You can’t put an underscore at the start of the literal.
- You can’t put an underscore at the end of the literal (including just before the suffix).

- You can't put an underscore directly before or after the period in a floating-point literal.
- In C# 7.0 and 7.1, you can't put an underscore after the base specifier (0x or 0b) of an integer literal.

The final restriction has been lifted in C# 7.2. Although readability is subjective, I definitely prefer to use an underscore after the base specifier when there are underscores elsewhere, as in the following examples:

- 0b_1000_0111 versus 0b1000_0111
- 0x_ffff_0000 versus 0xfffff_0000

That's it! A nice simple feature with very little nuance. The next feature is similarly straightforward and permits a simplification in some cases where you need to throw an exception conditionally.

14.4 Throw expressions

Earlier versions of C# always included the `throw` statement, but you couldn't use `throw` as an expression. Presumably, the reasoning was that you wouldn't want to, because it would always throw an exception. It turns out that as more language features were added that needed expressions, this classification became increasingly more irritating. In C# 7, you can use *throw expressions* in a limited set of contexts:

- As the body of a lambda expression
- As the body of an expression-bodied member
- As the second operand of the ?? operator
- As the second or third operand of the conditional ?: operator (but not both in the same expression)

All of these are valid:

```
public void UnimplementedMethod() =>
    throw new NotImplementedException();
```

Expression-bodied method


```
public void TestPredicateNeverCalledOnEmptySequence()
{
    int count = new string[0]
        .Count(x => throw new Exception("Bang!")); ←
    Assert.AreEqual(0, count);
}
```

Lambda expression


```
public static T CheckNotNull<T>(T value, string paramName) where T : class
    => value ??
        throw new ArgumentNullException(paramName);
```

?? operator (in expression-bodied method)


```
public static Name =>
    initialized
    ? data["name"]
    : throw new Exception("...");
```

: operator (in expression-bodied property)

You can't use `throw` expressions everywhere, though; that just wouldn't make sense. For example, you can't use them unconditionally in assignments or as method arguments:

```
int invalid = throw new Exception("This would make no sense");
Console.WriteLine(throw new Exception("Nor would this"));
```

The C# team has given us flexibility where it's useful (typically, where it allows you to express the exact same concepts as before, but in a more concise fashion) but prevented us from shooting ourselves in the foot with `throw` expressions that would be ludicrous in context.

Our next feature continues the theme of allowing us to express the same logic but with less fluff by simplifying the `default` operator with default literals.

14.5 Default literals (C# 7.1)

The `default(T)` operator was introduced in C# 2.0 primarily for use with generic types. For example, to retrieve a value from a list if the index was in bounds or the default for the type instead, you could write a method like this:

```
static T GetValueOrDefault<T>(IList<T> list, int index)
{
    return index >= 0 && index < list.Count ? list[index] : default(T);
```

The result of the `default` operator is the same default value for a type that you observe when you leave a field uninitialized: a null reference for reference types, an appropriately typed zero for all numeric types, U+0000 for `char`, `false` for `bool`, and a value with all fields set to the corresponding default for other value types.

When C# 4 introduced optional parameters, one way of specifying the default value for a parameter was to use the `default` operator. This can be unwieldy if the type name is long, because you end up with the type name in both the parameter type and its default value. One of the worst offenders for this is `CancellationToken`, particularly because the conventional name for a parameter of that type is `cancellationToken`. A common async method signature might be something like this:

```
public async Task<string> FetchValueAsync(
    string key,
    CancellationToken cancellationToken = default(CancellationToken))
```

The second parameter declaration is so long it needs a whole line to itself for book formatting; it's 64 characters.

In C# 7.1, in certain contexts, you can use `default` instead of `default(T)` and let the compiler figure out which type you intended. Although there are definitely benefits beyond the preceding example, I suspect it was one of the main motivating factors. The preceding example can become this:

```
public async Task<string> FetchValueAsync(
    string key, CancellationToken cancellationToken = default)
```

That's much cleaner. Without the type after it, `default` is a *literal* rather than an *operator*, and it works similarly to the `null` literal, except that it works for all types. The literal itself has no type, just like the `null` literal has no type, but it can be converted to any type. That type might be inferred from elsewhere, such as an implicitly typed array:

```
var intArray = new[] { default, 5 };
var stringArray = new[] { default, "text" };
```

That code snippet doesn't list any type names explicitly, but `intArray` is implicitly an `int[]` (with the `default` literal being converted to 0), and `stringArray` is implicitly a `string[]` (with the `default` literal being converted to a null reference). Just like the `null` literal, there does have to be some type involved to convert it to; you can't just ask the compiler to infer a type with no information:

```
var invalid = default;
var alsoInvalid = new[] { default };
```

The `default` literal is classified as a constant expression if the type it's converted to is a reference type or a primitive type. This allows you to use it in attributes if you want to.

One quirk to be aware of is that the term `default` has multiple meanings. It can mean the default value of a type or the default value of an optional parameter. The `default` literal always refers to the default value of the appropriate type. That could lead to some confusion if you use it as an argument for an optional parameter that has a different default value. Consider the following listing.

Listing 14.11. Specifying a default literal as a method argument

```
static void PrintValue(int value = 10)
{
    Console.WriteLine(value);
}

static void Main()
{
    PrintValue(default);           ↗ Method argument
}                                ↗ is default for int.
```

This prints 0, because that's the default value for `int`. The language is entirely consistent, but this code could cause confusion because of the different possible meanings of `default`. I'd try to avoid using the `default` literal in situations like this.

14.6 Nontrailing named arguments (C# 7.2)

Optional parameters and named arguments were introduced as complementary features in C# 4, and both had ordering requirements: optional parameters had to come after all required parameters (other than parameter arrays), and named arguments had to come after all positional arguments. Optional parameters haven't changed, but the C# team has noticed that often named arguments can be useful as tools for increasing clarity, even for arguments in the middle of an argument list. This is particularly

true when the argument is a literal (typically, a number, Boolean, literal, or null) where the context doesn't clarify the purpose of the value.

As an example, I've been writing samples for the BigQuery client library recently. When you upload a CSV file to BigQuery, you can specify a schema, let the server determine the schema, or fetch it from the table if that already exists. When writing the samples for the autodetection, I wanted to make it clear that you can pass a null reference for the `schema` parameter. Written in the simplest—but not clearest—form, it's not at all obvious what the null argument means:

```
client.UploadCsv(table, null, csvData, options);
```

Before C# 7.2, my options for making this clearer were to either use named arguments for the last three parameters, which ended up looking a little awkward, or use an explanatory local variable:

```
TableSchema schema = null;
client.UploadCsv(table, schema, csvData, options);
```

That's clearer, but it's still not great. C# 7.2 allows named arguments anywhere in the argument list, so I can make it clear what the second argument means without any extra statements:

```
client.UploadCsv(table, schema: null, csvData, options);
```

This can also help differentiate between overloads in some cases in which the argument (typically null) could be converted to the same parameter position in multiple overloads.

The rules for nontrailing named arguments have been designed carefully to avoid any subsequent positional arguments from becoming ambiguous: if there are any *unnamed* arguments after a named one, the named argument has to correspond to the same parameter as it would if it were a simple positional argument. For example, consider this method declaration and three calls to it:

```
void M(int x, int y, int z){}
M(5, z: 15, y: 10);      ← Valid: trailing named
                         arguments out of order
M(5, y: 10, 15);        ← Valid: nontrailing named
                         argument in order
M(y: 10, 5, 15);        ← Invalid: nontrailing named
                         argument out of order
```

The first call is valid because it consists of one positional argument followed by two named arguments; it's obvious that the positional argument corresponds to the parameter `x`, and the other two are named. No ambiguity.

The second call is valid because although there's a named argument with a later positional argument, the named argument corresponds to the same parameter as it would if it were positional (`y`). Again, it's clear what value each parameter should take.

The third call is invalid: the first argument is named but corresponds to the second parameter (`y`). Should the second argument correspond to the first parameter (`x`) on

the grounds that it's the first non-named argument? Although the rules could work this way, it all becomes a bit confusing; it's even worse when optional parameters get involved. It's simpler to prohibit it, so that's what the language team decided to do. Next is a feature that has been in the CLR forever but was exposed only in C# 7.2.

14.7 Private protected access (C# 7.2)

A few years ago, `private protected` was going to be part of C# 6 (and perhaps they planned to introduce it even earlier than this). The problem was coming up with a name. By the time the team had reached 7.2, they decided they weren't going to find a better name than `private protected`. This combination of access modifiers is more restrictive than either `protected` or `internal`. You have access to a `private protected` member only from code that's in the same assembly *and* is within a subclass of the member declaration (or is in the same type).

Compare this with `protected internal`, which is less restrictive than either `protected` or `internal`. You have access to a `protected internal` member from code that's in the same assembly *or* is within a subclass of the member declaration (or is in the same type).

That's all there is to say about it; it doesn't even merit an example. It's nice to have from a completeness perspective, as it was odd for there to be an access level that could be expressed in the CLR but not in C#. I've used it only once so far in my own code, and I don't expect it to be something I find much more useful in the future. We'll finish this chapter with a few odds and ends that don't fit in neatly anywhere else.

14.8 Minor improvements in C# 7.3

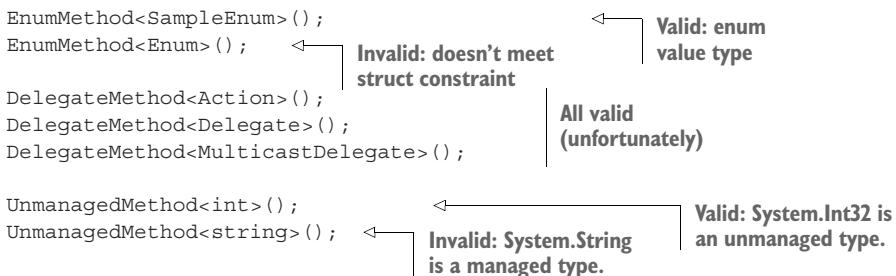
As you've already seen in this chapter and earlier in the book, the C# design team didn't stop work on C# 7 after releasing C# 7.0. Small tweaks were made, mostly to enhance the features released in C# 7.0. Where possible, I've included those details along with the general feature description. A few of the features in C# 7.3 don't fit in that way, and they don't really fit in with this chapter's theme of concise code, either. But it wouldn't feel right to leave them out.

14.8.1 Generic type constraints

When I briefly described type constraints in section 2.1.5, I left out a few restrictions. Prior to C# 7.3, a type constraint couldn't specify that the type argument must derive from `Enum` or `Delegate`. This restriction has been lifted, and a new kind of constraint has been added: a constraint of `unmanaged`. The following listing gives examples of how these constraints are specified and used.

Listing 14.12 New constraints in C# 7.3

```
enum SampleEnum {}  
static void EnumMethod<T>() where T : struct, Enum {}  
static void DelegateMethod<T>() where T : Delegate {}  
static void UnmanagedMethod<T>() where T : unmanaged {}  
...
```



I've shown a constraint of `where T : struct`, `Enum` for the enum constraint, because that's how you almost always want to use it. That constrains `T` to be a real enum type: a value type derived from `Enum`. The `struct` constraint excludes the `Enum` type itself. If you're trying to write a method that works with any enum type, you usually wouldn't want to handle `Enum`, which isn't really an enum type in itself. Unfortunately, it's far too late to add these constraints onto the various enum parsing methods in the framework.

The delegate constraint doesn't have an equivalent, unfortunately. There's no way of expressing a constraint of "only the types declared with a delegate declaration." You could use a constraint of `where T : MulticastDelegate` instead, but then you'd still be able to use `MulticastDelegate` itself as a type argument.

The final constraint is for *unmanaged* types. I've mentioned these in passing before, but an unmanaged type is a non-nullable, nongeneric value type whose fields aren't reference types, recursively. Most of the value types in the framework (`Int32`, `Double`, `Decimal`, `Guid`) are unmanaged types. As an example of a value type that isn't, a `ZonedDateTime` in Noda Time wouldn't be an unmanaged type because it contains a reference to a `DateTimeZone` instance.

14.8.2 Overload resolution improvements

The rules around overload resolution have been tweaked over and over again, usually in hard-to-explain ways, but the change in C# 7.3 is welcome and reasonably simple. A few conditions that used to be checked after overload resolution had finished are now checked earlier. Some calls that would have been considered to be ambiguous or invalid in an earlier version of C# are now fine. The checks are as follows:

- Generic type arguments must meet any constraints on the type parameters.
- Static methods can't be called as if they were instance methods.
- Instance methods can't be called as if they were static methods.

As an example of the first scenario, consider these overloads:

```

static void Method<T>(object x) where T : struct =>           ← Method with a
    Console.WriteLine($"{typeof(T)} is a struct");                  struct constraint

static void Method<T>(string x) where T : class =>             ← Method with a
    Console.WriteLine($"{typeof(T)} is a reference type");          class constraint
...
Method<int>("text");

```

In previous versions of C#, overload resolution would've ignored the type parameter constraints to start with. It would've picked the second overload, because `string` is a more specific regular parameter type than `object`, and then discovered that the supplied type argument (`int`) violated the type constraint.

With C# 7.3, the code compiles with no error or ambiguity because the type constraint is checked as part of finding applicable methods. The other checks are similar; the compiler discards methods that would be invalid for the call earlier than it used to. Examples of all three scenarios are in the downloadable source code.

14.8.3 **Attributes for fields backing automatically implemented properties**

Suppose you want a trivial property backed by a field, but you need to apply an attribute to the field to enable other infrastructure. Prior to C# 7.3, you'd have to declare the field separately and then write a simple property with boilerplate code. For example, suppose you wanted to apply a `DemoAttribute` (just an attribute I've made up) to a field backing a `string` property. You'd have needed code like this:

```
[Demo]  
private string name;  
public string Name  
{  
    get { return name; }  
    set { name = value; }  
}
```

That's annoying when automatically implemented properties do almost everything you want. In C# 7.3, you can specify a field attribute directly to an automatically implemented property:

```
[field: Demo]  
public string Name { get; set; }
```

This isn't a new modifier for attributes, but previously it wasn't available in this context. (At least not officially and not in the Microsoft compiler. The Mono compiler has allowed it for some time.) It's just another rough edge of the specification where the language wasn't consistent that has been smoothed out for C# 7.3.

Summary

- Local methods allow you to clearly express that a particular piece of code is an implementation detail of a single operation rather than being of general use within the type itself.
- `out` variables are pure ceremony reduction that allow some cases that involved multiple statements (declaring a variable and then using it) to be reduced to a single expression.

- Binary literals allow more clarity when you need to express an integer value, but the bit pattern is more important than the magnitude.
- Literals with many digits that could easily become confusing to the reader are clearer when digit separators are inserted.
- Like out variables, throw expressions often allow logic that previously had to be expressed in multiple statements to be represented in a single expression.
- Default literals remove redundancy. They also stop you from having to say the same thing twice.⁵
- Unlike the other features, using nontrailing named arguments may increase the size of your source code, but all in the name of clarity. Or, if you were previously specifying lots of named arguments when you wanted to name only one in the middle, you'll be able to remove some names without losing readability.

⁵ See how annoying redundancy is? Sorry, I couldn't resist.

15

C# 8 and beyond

This chapter covers

- Expressing null and non-null expectations for reference types
- Using switch expressions with pattern matching
- Matching patterns recursively against properties
- Using index and range syntax for concise and consistent code
- Using asynchronous versions of the `using`, `foreach`, and `yield` statements

At the time of this writing, C# 8 is still being designed. The GitHub repository shows a lot of potential features, but only a few have reached the stage of publicly available preview builds of the compiler. This chapter is educated guesswork; nothing here is set in stone. It's almost inconceivable that all the features being considered would be included in C# 8, and I've restricted myself to the ones I consider reasonably likely to make the cut. I've provided the most detail about the features available in preview at the time of writing, but even so, that doesn't mean further changes won't occur.

NOTE At the time of this writing, only a few C# 8 features are available in preview builds, and there are different builds with different features. The preview for nullable reference types supports only full .NET projects (rather than .NET Core SDK style projects), which makes it harder to experiment with them on real code if all your projects use the new project format. I expect these limitations to be overcome in later builds, possibly by the time you read this.

We'll start with nullable reference types.

15.1 Nullable reference types

Ah, null references. The so-called billion-dollar mistake that Tony Hoare apologized for in 2009 after introducing them in the 1960s. It's hard to find an experienced C# developer who hasn't been bitten by a `NullReferenceException` at least a few times. The C# team has a plan to tame null references, making it clearer where we should expect to find them.

15.1.1 What problem do nullable reference types solve?

As an example that I'll expand on over the course of this section, let's consider the classes in the following listing. If you're following along in the downloadable source code, you'll see that I've declared them as separate nested classes within each example, as the code changes over time.

Listing 15.1 Initial model before C# 8

```
public class Customer
{
    public string Name { get; set; }
    public Address Address { get; set; }
}

public class Address
{
    public string Country { get; set; }
}
```

An address would usually contain far more information than a country, but a single property is enough for the examples in this chapter. With those classes in place, how safe is this code?

```
Customer customer = ...;
Console.WriteLine(customer.Address.Country);
```

If you know (somehow) that `customer` is non-null and that a customer always has an associated address, that may be fine. But how can you know that? If you know that only because you've looked at documentation, what has to change to make the code safer?

Since C# 2, we've had nullable value types, non-nullable value types, and implicitly nullable reference types. A grid of nullable/non-nullable against value/reference

types has had three of the four cells filled in, but the fourth has been elusive, as shown in table 15.1.

Table 15.1 Support for nullability and non-nullability for reference and value types in C# 7

	Nullable	Non-nullable
Reference types	Implicit	Not supported
Value types	Nullable<T> or ? suffix	Default

The fact that there's only one supported cell in the top row means we have no way of expressing an intention that some reference values may be null and others should never be null. When you run into a problem with an unexpected null value, it can be hard to determine where the fault lies unless the code has been carefully documented with null checks implemented consistently.¹

Given the huge body of .NET code that now exists with no machine-readable discrimination between references that can reasonably be null and those that must always be non-null, any attempt to rectify this situation can only be a cautious one. What can we do?

15.1.2 *Changing the meaning when using reference types*

The broad idea of the null safety feature is to assume that when a developer is intentionally discriminating between non-null and nullable reference types, the default is to be non-nullable. New syntax is introduced for nullable reference types: `string` is a non-nullable reference type, and `string?` is a nullable reference type. The grid then evolves, as shown in table 15.2.

Table 15.2 Support for nullability and non-nullability for reference and value types in C# 8

	Nullable	Non-nullable
Reference types	No CLR type representation, but the ? suffix as an annotation	Default when nullable reference type support is enabled
Value types	Nullable<T> or ? suffix	Default

That sounds like the opposite of caution; it's changing the meaning of all C# code that deals with reference types! Turning on the feature changes the default from nullable to non-nullable. The expectation is that there are far fewer places where a null reference is intended to be valid than places where it should never occur.

Let's go back to our customer and address example. Without any changes to the code, the compiler warns us that our `Customer` and `Address` classes are allowing

¹ The day before I wrote this paragraph, most of my time was spent trying to track down a problem of exactly this kind. The issue is very real.

non-nullable properties to be uninitialized. That can be fixed by adding constructors with non-nullable parameters, as shown in the following listing.

Listing 15.2 Model with non-nullable properties everywhere

```
public class Customer
{
    public string Name { get; set; }
    public Address Address { get; set; }

    public Customer(string name, Address address) =>
        (Name, Address) = (name, address);
}

public class Address
{
    public string Country { get; set; }

    public Address(string country) =>
        Country = country;
}
```

At this point, you “can’t” construct a `Customer` without providing a non-null name and address, and you “can’t” construct an `Address` without providing a non-null country. I’ve deliberately put *can’t* in scare-quotes for reasons you’ll see in section 15.1.4.

But now consider our console output code again:

```
Customer customer = ...;
Console.WriteLine(customer.Address.Country);
```

This is safe, assuming everyone is obeying the contracts properly. Not only will it not throw an exception, but you won’t be passing a null value to `Console.WriteLine`, because the country in the address won’t be null.

Okay, so the compiler can check that things aren’t null. But what about when you want to allow null values? It’s time to explore the new syntax I mentioned before.

15.1.3 Enter nullable reference types

The syntax used to indicate a reference type that can be null is designed to be immediately familiar. It’s the same as the syntax for nullable value types: adding a question mark after the type name. This can be used in most places that a reference type can appear. For example, consider this method:

```
string FirstOrSecond(string? first, string second) =>
    first ?? second;
```

The signature of the method shows the following:

- The type of `first` is nullable string.
- The type of `second` is non-nullable string.
- The return type is non-nullable string.

The compiler then uses that information to warn you if try to misuse a value that might be null. For example, it can warn you if you do the following:

- Assign a possibly null value to a non-nullable variable or property.
- Pass a possibly null value as an argument for a non-nullable parameter.
- Dereference a possibly null value.

Let's build this into our customer model. Let's suppose the customer address could be null. You need to modify the `Customer` class as follows:

- Change the property type.
- Either remove the constructor parameter for the address, make it nullable, or overload it.

The `Address` type itself doesn't change, only how it's used. The following listing shows the new `Customer` class. I've chosen to remove the constructor parameter for the address.

Listing 15.3 Making the customer Address property nullable

```
public class Customer
{
    public string Name { get; set; }
    public Address? Address { get; set; }    ← The address is now
                                              optional information.

    public Customer(string name) =>      ← Removes the address parameter
        Name = name;                      from the constructor
}
```

Great, you've now made your intent clear: the `Name` property won't be null, but the `Address` property might be. The compiler now gives you a different warning when you try to display the country of the user's address:

```
CS8602 Possible dereference of a null reference.
```

Great! It's now identifying the problem you originally faced, which caused a `NullReferenceException`. How do you fix the problem? It's time to look at the *behavior* of nullable reference types rather than just the syntax.

15.1.4 Nullable reference types at compile time and execution time

One golden rule of the new feature is that no behavior is changed implicitly. Even though the meaning of your code has changed to assume an intent of non-nullable types, the behavior hasn't. The only difference is at compile time in terms of the warnings generated. No new real types are involved; the CLR has no notion of a nullable versus non-nullable reference type. Attributes are used to propagate nullability information, but that's all. This is similar to the extra information about tuple element names, which are not part of the type at execution time. This has two important consequences:

- Defensive programming remains a best practice. With the code you've written so far, it's possible for `Name` to be null, because a user could be ignoring warnings or using code from another project that uses only C# 7. Argument validation is still important.
- To understand the feature fully, you need to understand the compiler warnings. You definitely shouldn't just ignore them; they're present to provide value.

Let's look at the warning you're currently facing and consider all the ways you could avoid it. You currently have this:

```
Console.WriteLine(customer.Address.Country);
```

The compiler is correctly telling you this is dangerous because `customer.Address` could be null. You'll look at three ways you can make the code safer. First, you can use the null conditional and null coalescing operators in tandem, as shown in the next listing.

Listing 15.4 Safe dereferencing using the null conditional operator

```
Console.WriteLine(customer.Address?.Country ?? "(Address unknown)");
```

If `customer.Address` is null, the expression `customer.Address?.Country` won't try to evaluate the `Country` property, and the result of the expression will be null. The null coalescing operator then provides a default value to print. The compiler understands that you're no longer trying to dereference anything that might be null, and the warning goes away.

You may be a little uneasy with this at the moment. It's easy to get lost in a sea of question marks if you're not careful. I believe that C# developers will become more comfortable with this over time, but it's not the only solution available. You could take a more verbose approach that's simple to follow, as shown in the following listing.

Listing 15.5 Checking a reference with a local variable

```
Address? address = customer.Address;
if (address != null)
{
    Console.WriteLine(address.Country);
}
else
{
    Console.WriteLine("(Address unknown)");
}
```

Extracts address to a new local variable

Checks for nullity and dereferences only if non-null

There's an interesting point to note here: the compiler needs to keep track of more than just the type of the variable. If the rule were as simple as "dereferencing a value of a nullable reference type causes a warning," this code would still generate a warning, despite being safe. Instead, the compiler keeps track of whether a variable's value

can be null at each place in the code in a similar manner to the way it keeps track of definite assignment. By the time you reach the body of the `if` statement, the compiler knows that the value of `address` can't be null, so it doesn't warn when you dereference it. Our third approach, shown in the following listing, is similar to the second one, but without the local variable.

Listing 15.6 Checking a reference with repeated property access

```
if (customer.Address != null)
{
    Console.WriteLine(customer.Address.Country);
}
else
{
    Console.WriteLine("(Address unknown)");
}
```

Even when you understand how the second example compiles without a warning, listing 15.6 can be a little surprising. The compiler doesn't just keep track of whether a variable value can be null; it does that for properties, too. It assumes that if you access the same property on the same value twice, the result will be the same both times.

This may worry you. It means the feature isn't guaranteed to stop your code from dereferencing null values. Another thread could modify the `Address` property between the two calls you've seen, or the `Address` property itself could be written to randomly return a null value sometimes. There are other ways you can fool the compiler into believing your code is fine when it's not absolutely safe. This is known and accepted by the C# design team, because it's a pragmatic balance between safety and awkwardness. Code using the C# 8 features will be much more null-safe than code written before, but making it 100% safe would almost certainly require more-invasive changes that would put a lot of developers off. So long as you understand the limits of what it's trying to achieve, you'll be fine.

You've seen that the compiler works hard to understand what might or might not be null. What can you do when it doesn't have as much context as you do?

15.1.5 The damn it or bang operator

There's one additional piece of syntax you haven't looked at yet: the *dammit*, or *damn it*, or *bang operator*.² This is an exclamation mark at the end of an expression, and it's a way of telling the compiler to ignore whatever it thinks it knows about that expression and just treat it as non-null.

This is useful in two opposite situations:

- Sometimes you have more information than the compiler does, so you know a value won't be null, even though the compiler thinks it might be.

² I doubt that it'll ever officially be called the damn it operator, but I suspect the name will live on in the community, just like everyone calls the Microsoft .NET Compiler Platform by its original name of Roslyn.

- Sometimes you want to deliberately pass in a null value to check your argument validation.

Brief examples of the first situation are somewhat contrived, because you'd typically try to reorganize the code to avoid getting into that situation. In small examples, that's almost always feasible, but it's harder in real applications. The following listing shows a method to print the length of a string with input that can be null.

Listing 15.7 Using the bang operator to satisfy the compiler

```
static void PrintLength(string? text)           ← Input can be null
{
    if (!string.IsNullOrEmpty(text))           ← If IsNullOrEmpty returns
    {                                         false, it's not null.
        Console.WriteLine($"{text}: {text!.Length}"); ←
    }
    else                                     Use the bang operator to
    {                                         convince the compiler.
        Console.WriteLine("Empty or null");
    }
}
```

In this example, you know something the compiler doesn't in terms of the relationship between the input to `string.IsNullOrEmpty` and the return value. If `string.IsNullOrEmpty` returns `false`, the input can't be null, so it's fine to dereference that value to get the length of the string. If you just try to use `text.Length`, the compiler issues a warning. With `text!.Length`, you're telling the compiler that you know better, effectively taking responsibility for reasoning about the value.

Now it'd be nice if the compiler did understand that input/result relationship for `string.IsNullOrEmpty` method. We'll come back to that idea in section 15.1.7.

The second use of the bang operator is far easier to demonstrate with a realistic example. I mentioned earlier that you should still validate parameters for null, because it's still entirely possible for you to receive null values. You may then want to add a unit test for that validation, but then the compiler warns you because you're providing a null value when you've said it shouldn't be null. The following listing shows how the bang operator fixes this.

Listing 15.8 Using the bang operator in unit tests

```
public class Customer
{
    public string Name { get; }
    public Address? Address { get; }

    public Customer(string name, Address? address)
    {
        Name = name ?? throw new ArgumentNullException(nameof(name));
        Address = address;
    }
}
```

```

        }
    }

public class Address
{
    public string Country { get; }

    public Address(string country)
    {
        Country = country ???
            throw new ArgumentNullException(nameof(country));
    }
}

[TestMethod]
public void Customer_NameValidation()
{
    Address address = new Address("UK");
    Assert.Throws<ArgumentNullException>(
        () => new Customer(null!, address));    ← Deliberately passes in a null value
                                                for the non-nullable parameter
}

```

I've made the `Customer` and `Address` types immutable in listing 15.8 for simplicity. It's interesting to note that the compiler doesn't raise any kind of warning on the validation itself. Even though it knows the value shouldn't be null, it doesn't complain that the code checks whether it is null. But it does try to enforce that when you call the constructor in the test, the first argument is non-null. In an earlier version of C#, the lambda expression in the test would look like this:

```
(() => new Customer(null, address))
```

That code generates a warning, as you'd want it to in almost all cases. Changing the argument to `null!` satisfies the compiler, and the test does what you want. This raises the question of what it's like working with nullable reference types in practice, and, in particular, how to migrate existing code to use the feature.

15.1.6 Experiences of nullable reference type migration

There's no better way to get a feel for how a feature works than to try it. I used the C# 8 preview build with Noda Time to see how much work would be required to make it warning free and to see whether it found any bugs. This section describes this experience and some guidelines I found myself following. Your code may face different challenges, but I suspect there'll be plenty of commonality.

USING ATTRIBUTES TO EXPRESS NULLABLE INTENT BEFORE C# 8

For a long time, Noda Time has used attributes (at least for all public methods) to indicate whether reference type parameters can be null and likewise whether return values may return null. For example, here's the signature for a method in `IDateTimeZoneProvider`:

```
[CanBeNull] DateTimeZone GetZoneOrNull([NotNull] string id);
```

This shows that the argument for the `id` parameter must not be null, but the method may return a null reference. I've already expressed the intent around nullity, just not in a way that the C# compiler understood. That meant my first pass was just to go to all the places in the code where I'd said that null values were allowed and change them to use nullable reference types.

I happened to use the JetBrains annotations provided with ReSharper. This allows ReSharper to perform the same kind of inspection that C# 8 does in the language. I won't go into the details of these annotations other than to note that they're available. You don't have to use a third-party set of annotations at all, however. You can easily create your own attributes and apply them right now. Even without any tooling support, this can make your code easier to maintain, and you'll be in a better position to move to the C# 8 nullable reference types in the future.

ITERATION IS NATURAL

After this first pass, I had about 100 warnings. I went through and fixed most of those and then rebuilt. After the second pass, I had about 110 warnings—more than before! I went through and fixed most of those and then rebuilt. After the third pass, I still had about 100 warnings. I went through and fixed most of those and then rebuilt.

I don't remember how many iterations this took, but it's not a sign of anything being wrong. The process of making a codebase nullable-reference-type compliant is like playing whack-a-mole: you decide to change the nullability in one place, and then that can cause warnings everywhere that value is used. You change those, and the problem moves again. Decisions about nullability propagate through the code and need careful checking. This is fine and is what you should expect to happen.

But when part of the code needs a value to be nullable and another part needs it to be non-nullable, you've discovered a problem. This isn't a problem that C# 8 has introduced; it's a problem that the feature has revealed. How you handle it will be context specific.

BEST PRACTICES FOR USING THE BANG OPERATOR

If you have to use the bang operator in production code, add a comment to explain why you did so. If you use a nicely searchable format (for example, including `NULLABLEREF` in the comment), you'll be able to find them later. You may be able to remove the operator later through further tooling improvements. It's not that using the operator is wrong, but it's an assertion that you know better than the compiler, and I prefer not to trust myself that much.

I used the operator more often in test code and mostly for performing the sort of validation tests you saw in the previous section. Beyond that, if I expect a value to be non-null because of the way I've set up the test, I'm usually happy forcing the compiler to be happy with it, particularly if I know that it'll be validated by the code I'm calling afterward anyway. If I'm wrong, the result should be the test failing with either an `ArgumentNullException` or `NullReferenceException`, which is fine, as I'd still know that my assumptions were invalid. Arguably, test code should be less defensive

than production code in general; instead of trying to handle unexpected situations in a graceful way, it's fine for them to fail.

NULL-INCONSISTENT GENERICS

I found it odd to implement `IEqualityComparer<T>` for reference types in Noda Time, because it was defined long before nullable reference types were considered. Both `Equals` and `GetHashCode` are defined in terms of parameters of type `T`, but they're inconsistent in terms of null handling: `Equals` is meant to handle null values, but `GetHashCode` is meant to throw an `ArgumentNullException`.

It's unclear how this should be expressed in implementations. If I have an equality comparer for the `Period` class, should I implement `IEqualityComparer<Period?>` to allow null arguments or `IEqualityComparer<Period>` to prohibit them? Either way, callers could be surprised either at compile time or execution time.

Beyond just an implementation issue, it's unclear to me how this could be expressed more clearly in the interface itself. More language design work may be required here in order to express how generic type parameters should be handled. Just using `T?` in the interface would feel wrong, as you wouldn't want to accept `Nullable<T>` when `T` is a value type.

Although I happened to encounter this with `IEqualityComparer<T>`, I anticipate the same issue cropping up in other interfaces and even in generic classes. I'm mostly mentioning it here so that you don't think you've done anything wrong when you come across it.

THE END RESULT

The Noda Time codebase isn't huge, but it's not tiny either. The whole process took me about five hours, including time diagnosing a bug in the preview build of Roslyn. In the end, I found a bug (now fixed) in Noda Time around inconsistent handling of an odd situation where `TimeZoneInfo.Local` returns null in some environments on Mono. I also found some missing annotations and had to clarify the intent for some internal members.

I was pleased with the result; knowing the compiler was checking the consistency of the code improves my confidence in it. Additionally, after I've published a version of Noda Time built with C# 8, anyone using the library from C# 8 will benefit from the extra information. This will help move more errors from execution time to compile time, giving users more confidence in how they're using Noda Time. It's a win-win situation.

All of this experience was with the preview from the first half of 2018. This isn't the end state of the language design or the implementation, however. Let's take a speculative look at the future.

15.1.7 Future improvements

In June 2018, I spent time in conferences and user groups with Mads Torgersen, the lead of the C# language design team. I traveled with a laundry list of feature requests and issues based on my experience with Noda Time, and his responses reassured me about the future of the features.

The C# team is aware that the preview that's available already isn't quite ready for mainstream adoption. A few things need a bit more work, but the preview allows the team to gather early feedback. The changes listed here won't be the only ones, but they're the ones I was most interested in.

PROVIDING THE COMPILER WITH MORE SEMANTIC INFORMATION

When I introduced the bang operator in section 15.1.5, I showed that the compiler didn't understand the semantics of `string.IsNullOrEmpty`. (The compiler doesn't infer that if the method returns `false`, the input couldn't have been null.) This isn't the only situation in which a relationship between input and output should be able to help the compiler. Here are three examples that feel like they should compile without warnings (including `string.IsNullOrEmpty` again for completeness):

```
string? a = ...;
if (!string.IsNullOrEmpty(a))
{
    Console.WriteLine(a.Length);
}

object b = ...;
if (!ReferenceEquals(b, null))
{
    Console.WriteLine(b.GetHashCode());
}

XElement c = ...;
string d = (string) c;
```

In each case, the semantics of the code you're calling are important. For these examples, the compiler would need to know the following:

- If the result of `string.IsNullOrEmpty` is `false`, the input can't be null.
- If the result of `ReferenceEquals` is `false` and one of the inputs is known to be a null reference, the other input can't be null.
- If the input to the `XElement` to `string` conversion operator is non-null, the output is also non-null.

These are all examples of relationships between inputs and outputs, and those relationships can't be expressed at the moment. I suspect that most uses of the bang operator in the preview build could be avoided if the compiler understood these relationships. How can the compiler get that extra information?

One approach that could work for these specific examples would be for the compiler to have the information hardcoded. That would be easy for the C# design team but unsatisfactory in other ways. It'd put the framework libraries on a different footing to third-party libraries, which would be annoying. I may want to express relationships like this in Noda Time, for example, which would make it more pleasant to use.

It's likely that the C# team will instead design a whole new mini-language that can be expressed in attributes to give the compiler the extra semantic information it

needs to be smarter about determining whether a particular value should be considered “definitely not null.” This will require a lot of work to design and implement but will provide a much more complete solution.

DEEPER THINKING ABOUT GENERICS

Generics present interesting challenges for nullability design. I mentioned one example when implementing `IEqualityComparer<T>`, but the issue goes well beyond that. Consider the following simple class that’s already valid in C# 7:

```
public class Wrapper<T>
{
    public T Value { get; set; }
}
```

Should that be valid, and what does it mean? In particular, what’s the result of constructing an instance of it without setting the `Value` property?

- For `Wrapper<int>`, the value of `Value` will be 0 by default.
- For `Wrapper<int?>`, the value of `Value` will be the null value for `int?` by default.
- For `Wrapper<string>`, the value of `Value` will be a null reference by default. That’s bad, as it goes against the type of `Value` being the non-nullable string type.
- For `Wrapper<string?>`, the value of `Value` will be a null reference by default. That’s okay, as the type of `Value` is the nullable string type.

It gets even more confusing when you consider that at execution time, `Wrapper<int>` and `Wrapper<int?>` will be different CLR types, but `Wrapper<string>` and `Wrapper<string?>` will be the same CLR type.

I don’t know how this confusion will be resolved in C# 8, but the team is aware of it. I’m glad it’s their job rather than mine to make sense of it, as it makes my head hurt just thinking about it.

That example uses only syntax that’s valid in C# 7 and doesn’t explicitly refer to nullable types at all. What if you try to use `T?` within a generic type or method?

In C# 7, if you have a type parameter `T`, the type `T?` can be used only when `T` is constrained to be a non-nullable value type, at which point it means `Nullable<T>`. That’s reasonably simple, but what can you do for nullable reference types? It seems likely that you’ll need a new generic constraint of non-nullable reference type, at which point `T?` could be used when `T` is either constrained to be a non-nullable value type or is constrained to be a non-nullable reference type. I wouldn’t expect a single constraint to indicate “some non-nullable type,” because the representation of the corresponding nullable type is very different between value types and reference types.

OPT-IN PARAMETER VALIDATION

The only changes implemented so far have been at compile time. The IL generated by the compiler doesn’t change, and you still need to perform parameter validation to protect against code that ignores compiler warnings, uses the bang operator, or is compiled against an earlier version of C#.

That makes sense, but the validation feels like boilerplate code. The null-coalescing operator, nameof operator, and throw expressions are all features that have helped improve the code required for validation in some cases, but it's still annoying and easy to forget.

One feature under discussion is to allow an exclamation mark after a parameter name to indicate that the compiler should generate a null validation at the start of a method. Consider a method that might currently be written like this:

```
static void PrintLength(string text)
{
    string validated =
        text ?? throw new ArgumentNullException(nameof(text));
    Console.WriteLine(validated.Length);
}
```

You could instead write this:

```
static void PrintLength(string text!)
{
    Console.WriteLine(text.Length);
}
```



Automatic null validation

It's possible that properties could have automatic validation in the same way.

ENABLING NULLABILITY CHECKING

In the preview build I've used, nullability checking is turned on by default. Although you can suppress warnings in the normal way, it's likely that the C# 8 compiler will have more nuanced settings before it launches. There are lots of different scenarios to consider.

When developers upgrade to the C# 8 compiler, they're likely to want to do this without seeing any new warnings. This is particularly important if the project settings treat warnings as errors. I suspect this means nullability checking will be turned off by default, at least for existing projects.

Not all class libraries will embrace C# 8 at the same time. It'll be important for code that uses C# 8 with nullability checking turned on to be able to consume libraries that haven't migrated yet. This is likely to be geared toward reporting as few errors as possible. For example, the compiler could treat all inputs to the library as nullable but all outputs from the library as non-nullable. Additionally, there'll need to be a way for a library to indicate when it has migrated.

When developers decide to migrate a project to use nullable reference types, they may want to do so over the course of several changes. It's possible that their project may contain generated code that can't be easily modified to express nullability. This suggests it'd be useful to be able to express the concept of "this code expresses nullability" on a per type basis.

These considerations are new for C#. We've never had a language feature with such a broad impact on compatibility. I expect the team to iterate on this aspect several times before the final launch on C# 8.

Nullable reference types likely will be the biggest feature in C# 8, but others are also available in preview builds already. One of my favorites is switch expressions.

15.2 Switch expressions

The switch statement has been available in C# right from the start, and the only way it has changed in all that time is to permit pattern matching in C# 7. It remains an imperative control structure: if this case matches, do this; if that case matches, do that. A lot of the uses of switch statements are more functional, though, with each case computing a result: if this case matches, the result is X; if that case matches, the result is Y. This is a common construct in functional programming languages in which many functions are expressed purely in terms of pattern matching.

The introduction of expression-bodied members has made this stick out like a sore thumb. Many methods can be implemented with a single expression, but if you want to use a switch/case structure, you have to use a block body. This is usually just an inconvenience, but it's still a point of friction.

C# 8 introduces *switch expressions* as an alternative to switch statements. This uses somewhat different syntax from switch statements, so it's worth comparing the two. In chapter 12, when I introduced pattern matching, you looked at an example of a switch statement to compute the perimeter of different shapes. Here's the code used in chapter 12:

```
static double Perimeter(Shape shape)
{
    switch (shape)
    {
        case null:
            throw new ArgumentNullException(nameof(shape));
        case Rectangle rect:
            return 2 * (rect.Height + rect.Width);
        case Circle circle:
            return 2 * PI * circle.Radius;
        case Triangle triangle:
            return triangle.SideA + triangle.SideB + triangle.SideC;
        default:
            throw new ArgumentException(
                $"Shape type {shape.GetType()} perimeter unknown",
                nameof(shape));
    }
}
```

The following listing shows the equivalent code using a switch expression instead but still using a regular block-bodied method.

Listing 15.9 Converting a switch statement into a switch expression

```
static double Perimeter(Shape shape)
{
    return shape switch
    {
```

```

        null => throw new ArgumentNullException(nameof(shape)),
        Rectangle rect => 2 * (rect.Height + rect.Width),
        Circle circle => 2 * PI * circle.Radius,
        Triangle triangle =>
            triangle.SideA + triangle.SideB + triangle.SideC,
        _ => throw new ArgumentException(
            $"Shape type {shape.GetType()} perimeter unknown",
            nameof(shape))
    };
}

```

There are a lot of things to point out here, so I haven't tried to cram them all into the code as annotations. Here are all the differences between a switch statement and a switch expression:

- Instead of `switch (value)`, the introductory syntax for switch expressions is `value switch`.
- A fat arrow `=>` comes between the pattern and the result to return if that pattern is matched. (In a switch statement, a colon is used instead.)
- The `case` keyword isn't used at all in switch expressions. The left side of the `=>` is just a pattern with an optional guard clause with the `when` keyword.
- The right side of the `=>` is just an expression. The `return` keyword isn't used, because every pattern results in a value or throws. Likewise, there's never a `break` statement.
- The patterns are comma separated. If you're converting a switch statement into a switch expression, this usually means changing semicolons into commas.
- There's no default case. Instead, the discard `_` (underscore) is used to match anything that hasn't already been matched.

My experience has mostly been writing methods that return a switch expression result directly, but you can also use it like any other expression. For example, you could write this:

```

double circumference = shape switch
{
    <--> Body of switch
    expression as before
};

```

This is fine, but as I mentioned before, one of the nicest aspects of switch expressions is to use them for expression-bodied methods. The following listing shows the evolution of listing 15.9 into an expression-bodied method.

Listing 15.10 Using a switch expression to implement an expression-bodied method

```

static double Perimeter(Shape shape) =>
    shape switch
    {
        null => throw new ArgumentNullException(nameof(shape)),
        Rectangle rect => 2 * (rect.Height + rect.Width),

```

```

Circle circle => 2 * PI * circle.Radius,
Triangle triangle =>
    triangle.SideA + triangle.SideB + triangle.SideC,
_ => throw new ArgumentException(
    $"Shape type {shape.GetType()} perimeter unknown",
    nameof(shape))
};

```

You can format this however you like, perhaps moving the `shape` switch onto the first line, or maybe outdenting the braces to the same level as the method declaration.

One important difference between switch statements and switch expressions is that there must always be some result (which could be an exception) from a switch expression. A switch expression isn't allowed to do nothing and produce no value. You can use the `_` discard to make sure of that, but it's possible to write a switch expression that isn't *exhaustive*—in other words, an expression that may not always match. With the preview build I've been working with, this produces a compiler warning, and then the compiler emits invalid IL. This might become a compile-time error instead, or the compiler may inject code to throw an exception (possibly `InvalidOperationException`) to indicate that the code encountered a situation it didn't expect.

The one issue I have with switch expressions at the moment is that there's no way of expressing multiple patterns that should evaluate to the same result. In a switch statement, you can specify multiple case labels, but there's no equivalent in switch expressions yet. The C# team is aware of the desire for this, so hopefully it will be included before C# 8 is released.

The use of patterns in C# 8 isn't just improved via switch expressions. The patterns themselves are growing in scope.

15.3 Recursive pattern matching

As a reminder, the patterns introduced in C# 7 were as follows:

- Type patterns (`expression is Type t`)
- Constant patterns (`expression is 10, expression is null, and so on`)
- The `var` pattern (`expression is var v`)

C# 8 will introduce recursive patterns (patterns can be nested within bigger patterns) as well as deconstruction patterns. The simplest way of explaining recursive patterns is to show them in action. We'll come back to deconstruction patterns.

15.3.1 Matching properties in patterns

To match properties with additional patterns inside an overall pattern, you use braces containing a comma-separated list of patterns against properties. The property patterns match the property value against the nested pattern using any of the normal pattern types. As an example, let's have another look at the three patterns we're using to work out the areas of rectangles, circles, and triangles taken from listing 15.10:

```

Rectangle rect => 2 * (rect.Height + rect.Width),
Circle circle => 2 * PI * circle.Radius,
Triangle triangle => triangle.SideA + triangle.SideB + triangle.SideC,

```

In each case, you don't need the shape itself; you just need properties from it. You can use nested var patterns to match those properties against any value and extract pattern variables for each of the properties you need. The following listing shows the full method with the nested patterns.

Listing 15.11 Matching nested patterns

```
static double Perimeter(Shape shape) => shape switch
{
    null => throw new ArgumentNullException(nameof(shape)),
    Rectangle { Height: var h, Width: var w } => 2 * (h + w),
    Circle { Radius: var r } => 2 * PI * r,
    Triangle { SideA: var a, SideB: var b, SideC: var c } => a + b + c,
    _ => throw new ArgumentException(
        $"Shape type {shape.GetType()} perimeter unknown", nameof(shape))
};
```

Is this clearer than the previous code? I'm not sure. I've used it as an example that follows neatly from the previous one, but I might easily stick with the code in listing 15.10. You'll look at a more complicated example later, in which the feature becomes more compelling but would be harder to immediately understand.

Note that although here you've stopped capturing the `Rectangle`, `Circle`, or `Triangle` in their own pattern variables (`rect`, `circle`, and `triangle` before), that's only because you don't need them for anything. It's still valid to introduce a pattern variable that way. For example, if you were describing shapes, you might have a pattern to describe a flat rectangle with zero height:

```
Rectangle { Height: 0 } rect => $"Flat rectangle of width {rect.Width}"
```

This is useful when you have a lot of properties but you're just testing patterns against a few of them. Next up, we'll look at *deconstruction patterns*.

15.3.2 Deconstruction patterns

You saw deconstruction of tuples in section 12.1 and deconstruction via the `Deconstruct` method in section 12.2. Patterns in C# 8 will be extended to allow deconstruction with nested patterns inside. As a somewhat contrived example, you might decide that it's natural to deconstruct a `Triangle` to all three of its sides:

```
public void Deconstruct
    (out double sideA, out double sideB, out double sideC) =>
    (sideA, sideB, sideC) = (SideA, SideB, SideC);
```

You could then simplify our perimeter computation to deconstruct to three variables instead of specifying each property name. So instead of this case in our switch expression

```
Triangle { SideA: var a, SideB: var b, SideC: var c } => a + b + c
```

you could have this:

```
Triangle (var a, var b, var c) => a + b + c
```

Again, is that more readable than just matching against the type? Maybe. Over time, I suspect each developer will work out their own preferences around pattern matching and ideally come to a convention within the codebases they're working in, too.

15.3.3 Omitting types from patterns

The ability to look inside objects makes patterns useful even when you're not testing the value's type. At that point, it feels redundant to specify the type as part of the pattern. For this example, let's go back to the customer and address example used for nullable reference types. You'll go back to the first data model: all mutable, all nullable:

```
public class Customer
{
    public string Name { get; set; }
    public Address Address { get; set; }
}

public class Address
{
    public string Country { get; set; }
}
```

Now suppose you want to greet customers in different ways depending on the country in their address. Your input could be of type `Customer`, so you don't want to have to repeat that within the pattern. When you match the `Address` of a customer within a pattern, that will always be of type `Address`, so you don't need to specify that type either.

The following listing shows multiple patterns matching different kinds of customers. It also demonstrates the `{ }` pattern, which is a special case of a property pattern that doesn't have any properties to match. That pattern matches any non-null value.

Listing 15.12 Matching customers against multiple patterns concisely

```
static void Greet(Customer customer)
{
    string greeting = customer switch
    {
        { Address: { Country: "UK" } } => "Welcome, customer from the United Kingdom!",
        { Address: { Country: "USA" } } => "Welcome, customer from the USA!",
        { Address: { Country: string country } } => $"Welcome, customer from {country}!",
        { Address: { } } => "Welcome, customer whose address has no country!",
        { } => "Welcome, customer with a null address"
    };
}
```

Annotations:

- Matches a country of UK
- Matches a country of USA
- Matches any country, but it must be present
- Matches any address
- Matches any customer, even with a null address

```

    "Welcome, customer of an unknown address!",
    - => "Welcome, nullness my old friend!"
};

Console.WriteLine(greeting);
}

```

Matches anything, even a null customer reference

The ordering is important here. For example, a customer with an address with a country of USA could match every pattern except the first one. You could make the patterns more selective instead (using the constant null pattern to match customers with a null Address property value, for example), but it's simpler to rely on the ordering.

The enhancements to pattern matching in C# 8 will allow them to be used in more cases where currently you need `if` statements. Switch expressions add to this flexibility, too. I expect more and more code to be written with patterns. As always, it's important to avoid going over the top; not all code will be simpler when written with patterns than with the control structures we had before. Still, this area of C#'s evolution definitely has a lot of potential. Our next feature is really a pair of features enabled by two new framework types.

15.4 Indexes and ranges

Compared with nullable reference types and improved pattern handling, indexes and ranges feel like a small feature, even combined. But I suspect over time we'll come to wonder why it took so long to have them. The following listing provides a tiny taste before you look at the details.

Listing 15.13 Trimming the first and last character from a string with a range

```

string quotedText = "'This text was in quotes'";
Console.WriteLine(quotedText);
Console.WriteLine(quotedText.Substring(1..^1));

```

Takes a substring of the string with a range literal

The output is as follows:

```
'This text was in quotes'
This text was in quotes
```

The highlighted expression of `1..^1` is the interesting part here. To understand this code, you need to learn about two new types.

15.4.1 Index and Range types and literals

The idea is simple. `Index` and `Range` are two structs that will be provided in the framework but currently need to be defined in your own code:

- `Index` is an integer from either the start or end of something indexable. The value of the index is never negative.
- `Range` is a pair of indexes: one for the start of the range and one for the end.

There are then three pieces of important syntax:

- A regular implicit conversion from `int` to create a “from the start” `Index`.
- A new unary operator (`^`) that can be used with `int` to create a “from the end” `Index`. Here a value of 0 means the element just past the end, and a value of 1 means the last element.³
- A new binary-ish operator (`..`) with optional operands for the start and end to create a `Range`.

The `..` operator is binary-ish because there can be zero, one, or two operands. The following listing shows examples of all of these. You’re not applying the indexes or ranges to anything; you’re just creating the values.

Listing 15.14 Index and range literals

```
Index start = 2;
Index end = ^2;
Range all = ..;
Range startOnly = start..;
Range endOnly = ..end;
Range startAndEnd = start..end;
Range implicitIndexes = 1..5;
```

One point to note is that the start and end points of a range can be any index. For example, you could have a range of `^5..10` representing the fifth element from the end to the tenth element from the start. This would be unusual, but valid.

This is the sum total of the direct language support for indexes and ranges. It’s when they also have framework support that they become useful.

15.4.2 Applying indexes and ranges

All the examples in this section require extension methods and extension operators supported by the C# 8 preview build. The exact APIs may change, and the extensions provided in the preview work with only a limited set of types; this is just enough to demonstrate the benefits. In listing 15.13, I showed how the `Substring` method can be used with a `Range`. Both indexes and ranges will be applied and most often to types that represent sequences of some form, such as

- Arrays
- Spans
- Strings (as sequences of UTF-16 code units)

These all support two operations:

- Retrieving a single element
- Creating a slice to represent part of the sequence

³ This is slightly counterintuitive when using an `Index` with an indexer, but it makes a lot more sense with ranges, which have exclusive upper bounds. A range with an upper bound of `^0` is effectively “to the end of the sequence,” which is probably what you’d expect.

The single-element-retrieval operation already has a common representation using an indexer accepting an `int` parameter, but this makes it hard to retrieve the last element in a uniform way. The `Index` type solves this with its from the start or from the end aspect. The slice operation has previously taken different forms depending on the type involved. For example, `Span<T>` has a `Slice` method, whereas `String` has a `Substring` method.

By adding indexer overloads accepting `Index` and `Range` values, you can use a consistent and convenient syntax to perform both operations on all of the relevant types. The following listing shows similar calls working for a string and a `Span<int>`.

Listing 15.15 Using indexer overloads for index and range in a string and a span

```

Accesses a single character
by index from start
string text = "hello world";
Console.WriteLine(text[2]);
Console.WriteLine(text[^3]);
Console.WriteLine(text[2..7])

Accesses a single character by
index from end
Takes a substring
using a range
Span<int> span = stackalloc int[] { 5, 2, 7, 8, 2, 4, 3 };
Console.WriteLine(span[2]);
Console.WriteLine(span[^3]);
Span<int> slice = span[2..7];
Console.WriteLine(string.Join(", ", slice.ToArray()));

Accesses a single element
by index from start
Accesses a single element by index from end
Creates a slice
using a range

```

The output is as follows:

```

1
r
llo w
7
2
7, 8, 2, 4, 3

```

Both the string and span indexers accepting a `Range` treat the upper bound of the range as exclusive: the range `[2..7]` returns the elements with indexes 2, 3, 4, 5, and 6.

In listing 15.15, the ranges included both start and end indexes, and both index values were computed from the start. You can use any range with the indexers so long as the indexes are valid for the sequence they're applied to. For example, using `text[^5..]` with the code in listing 15.15 would return `world` as the last five characters of `text`.

Likewise, you could write `text[^10..5]`, which would return `ello`. In the context of a string of length 11 (`hello world`), an index of `^10` is equivalent to an index of 1, so `text[^10..5]` is equivalent (in this case, it does depend on the length of `text`) to `text[1..5]`, returning the four characters after the first. Next, we'll look at increased language support for asynchrony.

15.5 More async integration

When `async/await` was introduced in C# 5, it revolutionized asynchrony for many C# developers. But a few language features have so far stayed synchronous, making it hard to go all in on asynchrony. In this section, we'll look at the following:

- Async disposal
- Async iteration (`foreach`)
- Async iterators (`yield return`)

These require framework support as well as language support. It wouldn't be appropriate for the compiler to approximate asynchrony by executing the synchronous code on a different thread, for example. Let's start with `async` disposal, which is the simplest of the three features.

15.5.1 Asynchronous resource disposal with using await

The `IDisposable` interface with its single `Dispose` method is naturally synchronous. If that method needs to perform I/O, such as to flush a stream, then it can block with all the normal issues that causes.

A new interface will be introduced for classes that support asynchronous disposal:

```
public interface IAsyncDisposable
{
    Task DisposeAsync();
}
```

There's no requirement that a type that implements `IAsyncDisposable` also implements `IDisposable`, although I suspect many types will do so.

There's then corresponding language support in the form of the `using await` statement, which works as you'd expect it to, calling `DisposeAsync` automatically and awaiting the resulting task. The following listing shows an example of implementing the interface and then using it.

Listing 15.16 Implementing `IAsyncDisposable` and calling it with `using await`

```
class AsyncResource : IAsyncDisposable
{
    public async Task DisposeAsync()
    {
        Console.WriteLine("Disposing asynchronously...");
        await Task.Delay(2000);
        Console.WriteLine("... done");
    }

    public async Task PerformWorkAsync()
    {
        Console.WriteLine("Performing work asynchronously...");
        await Task.Delay(2000);
        Console.WriteLine("... done");
    }
}
```

```
async static Task Main()
{
    using await (var resource = new AsyncResource())
    {
        await resource.PerformWorkAsync();
    }
    Console.WriteLine("After the using await statement");
}
```

The output shows the resource disposal:

```
Performing work asynchronously...
... done
Disposing asynchronously...
... done
After the using await statement
```

This is simple, but it hides two aspects of complexity that need to be addressed:

- Libraries typically await tasks with `ConfigureAwait(false)`. Applications typically await tasks without this. If the compiler is doing the awaiting automatically, how can the user configure this?
- It'd be natural to have cancellation available for disposal. Where does that fit into the interface and the call site?

The C# team is aware of both points, and I expect them to be addressed in some form before release. The same problems occur for the other async features in C# 8, and I hope they'll all be solved in a similar way. Let's look at the next feature now: asynchronous iteration with `foreach`.

15.5.2 Asynchronous iteration with `foreach await`

Spoiler alert: there's quite a lot of text before we reach the language feature in this section. That's necessary in order to explain it properly, but the upshot is that code like this will be valid, where `asyncSequence` requires asynchronous work to retrieve the items:

```
foreach await (var item in asyncSequence)
{
    ↪ Uses item
}
```

The interfaces introduced for asynchronous iteration aren't quite as straightforward as the one for disposal. There are two interfaces, mirroring `IEnumerable<T>` and `IEnumerator<T>` to some extent, but not quite so obviously:

```
public interface IAsyncEnumerable<out T>
{
    IAsyncEnumerator<T> GetAsyncEnumerator();
}

public interface IAsyncEnumerator<out T>
{
```

```

    Task<bool> WaitForNextAsync() ;
    T TryGetNext(out bool success) ;
}

```

`IAsyncEnumerable<T>` may be closer to `IEnumerable<T>` than you expect; there's nothing asynchronous in it. Instead of `GetEnumerator()`, it has `GetAsyncEnumerator()`, and that returns an `IAsyncEnumerator<T>`, but it does so synchronously. It's possible that for some implementations this will be problematic, but I expect it to be the natural approach for most asynchronous sequences. Any implementation that wants to perform asynchronous operations as part of setup will probably need to defer that work until the caller starts iterating over the result.

The `IAsyncEnumerator<T>` interface is much further from `IEnumerator<T>` and reflects a common pattern in real-world implementations. Asynchrony is often used when I/O is involved, such as retrieving results over a network. That often naturally results in sequences being retrieved in chunks; you may perform a query and retrieve the first 10 results together, then the next 7, and then be told that's the complete result set.

While you're iterating within a set of results that has been buffered, there's no need for asynchrony. Although asynchrony is quite efficient, it's not completely free, so it's worth avoiding if you can. Instead, you can iterate synchronously, so long as you have a way of determining when you've reached the end of the current result set. At that point, you can asynchronously fetch the next one and iterate through that synchronously again.

The `IAsyncEnumerator<T>` interface exposes this pattern through its two methods:

- `WaitForNextAsync` is asynchronous, returning a task that indicates whether any more results were retrieved or whether you've reached the end of the sequence.
- `TryGetNext` is synchronous, returning the next item. The `out` parameter is used to indicate whether there *was* a next item to return.⁴ When this is `false`, that doesn't mean you've necessarily reached the end of sequence; it just means you need to call `WaitForNextAsync` again.

That may all sound complicated, but the good news is that you're unlikely to need to do any of this yourself; the new `foreach await` statement handles it all for you.

Let's look at an example, which draws heavily from my experience working with Google Cloud Platform APIs. Many APIs have list operations, such as listing contacts in an address book or virtual machines in a cluster. There may be too many results to return in a single RPC response, so we have a page-based pattern: each response contains a "next page token" that the client supplies on a subsequent request to retrieve more data. For the first request, the client doesn't supply a page token, and the final

⁴ This is oddly inconsistent with most `TryXYZ` methods, which return `bool` and use an `out` parameter for the value. This could change before release.

response doesn't contain a page token. A simplified view of the API might look like the following listing.

Listing 15.17 Simplified RPC-based service for listing cities

```
public interface IGeoService
{
    Task<ListCitiesResponse> ListCitiesAsync(ListCitiesRequest request);
}

public class ListCitiesRequest
{
    public string PageToken { get; }
    public ListCitiesRequest(string pageToken) =>
        PageToken = pageToken;
}

public class ListCitiesResponse
{
    public string NextPageToken { get; }
    public List<string> Cities { get; }

    public ListCitiesResponse(string nextPageToken, List<string> cities) =>
        (NextPageToken, Cities) = (nextPageToken, cities);
}
```

That's unwieldy to use directly, but it can easily be wrapped in a client that exposes this API instead, as shown in the next listing.

Listing 15.18 Wrapper around the RPC service to provide a simpler API

```
public class GeoClient
{
    public GeoClient(IGeoService service) { ... }           ← Constructs a GeoClient
    public IAsyncEnumerable<string> ListCitiesAsync() { ... } ← with an RPC service
}
```

Provides a simple async sequence of cities

With `GeoClient` in place, you can finally use `foreach await`, as in the following listing.

Listing 15.19 Using foreach await with a GeoClient

```
var client = new GeoClient(service);

foreach await (var city in client.ListCitiesAsync())
{
    Console.WriteLine(city);
}
```

The final code here is a lot simpler than all the code I had to show you to set up the example, and that's without even looking at the implementation of `GeoClient`. But

that's a good thing; it shows the benefit of the feature. You've taken relatively complex definitions in both `IGeoService` and `IAsyncEnumerable<T>` and consumed them in a simple and efficient manner with `foreach await`.

NOTE The downloadable source code contains a complete example with an in-memory fake service implementation.

One thing you may be surprised about is that `IAsyncEnumerable<T>` doesn't implement `IAsyncDisposable`. That could change before release, but even if it doesn't, I expect the compiler to dispose of an enumerator if it turns out to implement `IAsyncDisposable` at execution time.

Just like the synchronous `foreach` statement, `foreach await` won't require the `IAsyncEnumerable<T>` and `IAsyncEnumerator<T>` interfaces to be implemented. It'll be pattern based, so any type providing a `GetAsyncEnumerator()` method that returns a type that in turn provides the appropriate `WaitForNextAsync` and `TryGetNext` methods will be supported. This could allow some optimizations, but I expect the interfaces to be used most of the time.

So far, you've seen how to consume asynchronous sequences. What about producing them?

15.5.3 Asynchronous iterators

C# 2 introduced iterators with `yield return` and `yield break` statements to make it easy to write methods returning `IEnumerable<T>` or `IEnumerator<T>`. C# 8 will have the same feature for asynchronous sequences. The feature isn't available in the preview, but the following listing shows how I expect it to work.

Listing 15.20 Implementing `ListCitiesAsync` with an iterator

```
public async IAsyncEnumerable<string> ListCitiesAsync()
{
    string pageToken = null;
    do
    {
        var request = new ListCitiesRequest(pageToken);
        var response = await service.ListCitiesAsync(request);
        foreach (var city in response.Cities)
        {
            yield return city;
        }
        pageToken = response.NextPageToken;
    } while (pageToken != null);
}
```

The mapping between the async iterator method and the `IAsyncEnumerable<T>` interface, with its mixture of asynchronous and synchronous parts, will be complex to implement. Whenever you continue executing code in the `async` method, it can complete that specific call in several ways:

- It could await an incomplete asynchronous operation.
- It could reach a `yield return` statement.
- It could reach a `yield break` statement.
- It could reach the end of the method.
- It could throw an exception.

How those are handled will depend on whether the caller is executing `WaitForNextAsync()` or `TryGetNext()`. To make this efficient, the generated code should effectively switch between synchronous mode (if you’re yielding values with no intervening awaits) and asynchronous mode (if you’re awaiting an asynchronous operation). I can broadly picture how this might be achieved, but I’m glad I’m not the one having to implement it.

There are other features not available in the C# 8 preview yet. We’ll look at these more briefly.

15.6 Features not yet in preview

If C# 8 turns out to have only the features I’ve listed so far, it’ll still be a big deal. In some ways, I wish we could have a release with just nullable reference types, wait a year or so for most codebases to be updated to it, and then continue with more features. But C# 8 likely will ship with more features than I’ve shown so far.

This section discusses the features I think are the most likely to be included in C# 8. Even more features have been proposed either by members of the C# team or by external developers. The C# team uses GitHub to keep track of language proposals, which makes it easy to see what’s going on and contribute yourself; see <https://github.com/dotnet/csharplang>. We’ll start with a feature inspired by Java.

15.6.1 Default interface methods

Whereas C# introduced extension methods for LINQ, Java took a different approach to enable its support for *streams*, which covers many of the same use cases as LINQ. In Java 8, Oracle introduced *default methods* in Java interfaces: an interface could declare a method and a default implementation for it, which could then be overridden within a concrete implementation. The default implementation can’t declare any state in terms of fields; it has to be expressed in terms of the other members of the interface.

The two features are similar in some ways: they both allow logic to be expressed so the consumer of an interface can call a method without every interface implementation having to directly know about it or implement it. There are pros and cons with each approach:

- Extension methods can be introduced by anyone, not just the author of the interface. You can’t add a default method to an interface you can’t control. (Extension methods can also be applied to classes and structs, of course.)
- Default methods can be overridden by implementing classes, often for the sake of optimization. Extension methods can’t be overridden; they’re just static

methods with syntactic sugar to make calling them look more like they're regular instance methods.

The second point can be easily appreciated using LINQ's `Enumerable.Count()` method as an example. By default, it counts the elements in a sequence by calling `GetEnumerator()` and then counting how many calls to `MoveNext()` on that enumerator return `true`.

Many implementations of `IEnumerable<T>` have far more efficient ways of determining the number of elements. `Enumerable.Count()` is specifically optimized for some of those, such as `ICollection` and `ICollection<T>` implementations. But what about a collection that doesn't want to implement either of those interfaces but still wants to provide the `Count` cheaply? It's stuck; it has no way of communicating to `Enumerable.Count()` that it can implement that part of LINQ itself more efficiently. If `Count()` had been a method in `IEnumerable<T>` with a default implementation, however, our new collection could just override that method.

Here's an example of how `IEnumerable<T>` could've been declared using C# 8 default interface methods:

```
public interface IEnumerable<T>
{
    I IEnumerator<T> GetEnumerator();

    int Count()
    {
        using (var iterator = GetEnumerator())
        {
            int count = 0;
            while (iterator.MoveNext())
            {
                count++;
            }
        }
        return count;
    }
}
```

Default interface methods also allow interfaces to be expanded over time in a rather more version-friendly way. New methods can be added with a default implementation that either implements the new functionality using the existing members or potentially throws a `NotSupportedException`. That way, old implementations will still build, even if the new method can't be called reliably. Versioning is a tricky subject, to say the least, but having another option in our toolbox is welcome. In numerous situations, this would've made things simpler in code that I maintain.

Default interface methods are proving to be a controversial feature. They require CLR support, which makes the feature harder to experiment with before committing to it wholeheartedly. If the feature is included, it'll be interesting to see its adoption rate. It may remain rarely used until the runtime versions that support it are widely

adopted, too. Next, we'll look at a feature that has been talked about and even prototyped for a long time.

15.6.2 Record types

The forerunner of record types was a feature called *primary constructors*, which was originally intended to be present in C# 6. The language team wasn't happy with some of the rough edges in the original design, so they decided to delay its introduction until it could be improved.

Record types are designed to make it easy to create immutable classes or structs with a given set of properties. I tend to think of them in terms of starting with anonymous types but adding all kinds of features. They can be declared incredibly simply. For example, here's a complete class declaration:

```
public class Point(int X, int Y, int Z);
```

That generates a bunch of members for you, although you can still introduce your own behavior as well. The generated members are a constructor, properties, equality methods, a `Deconstruct` method for deconstruction, and a `With` method like this:

```
public Point With(int X = this.X, int Y = this.Y, int Z = this.Z) =>
    new Point(X, Y, Z);
```

That isn't valid syntax for optional parameter default values at the moment, and it's not clear whether it'll be valid to write that code explicitly, but it at least shows the intention of the method's behavior.

The `With` method is designed to interoperate with new syntax in the form of *with expressions*. The idea is that both the method and the syntax make it easy to create a new instance of the immutable type that's the same as an existing one but with one or more properties changed. `WithFoo` methods are common in immutable types already (where `Foo` is the name of a property in the type), but they typically work on one property at a time. For example, with an immutable `Point` class with `X`, `Y`, and `Z` properties, you might use the following code to create a new point that has the same `Z` value as a previous point, but new `X` and `Y` values:

```
var newPoint = oldPoint.WithX(10).WithY(20);
```

Each `WithFoo` method calls a constructor, passing in all the existing properties other than the one named in the method, where the new value specified in the parameter is used. These methods become tedious to write and have a performance implication, too: to "change" `N` properties, you need to make `N` method calls, each one of which creates a new object.

The `With` method for record types is different: it has one parameter for each property of the type, with new syntax for a default parameter value if that parameter isn't specified, indicating that the value should be taken from the current object. For example, consider the `With` method in our `Point` type. You could either call that directly

```
var newPoint = oldPoint.With(X: 10, Y: 20);
```

or use the new *with expression* syntax, which looks more like an object initializer:

```
var newPoint = oldPoint with { X = 10, Y = 20 };
```

The two would compile to the same IL. This way, only a single new object is constructed.

This is only a simple example. It becomes trickier when you have a complex type and you want to modify just one leaf node. For example, if you have a `Contact` type with an `Address` property, you may want to create a new contact that's the same as the old one but with one part of the `Address` property different. It's possible that'll still be tricky in C# 8 but that *with expression* syntax may be enhanced to make that simpler over time, just as the syntax for pattern matching has grown.

I'm excited about the possibilities here. Immutable types have been a pain to create and work with in C# for a long time. Whereas C# 7 tuples filled one gap left by anonymous types, record types fill another. I've always loved anonymous types for the work the compiler does for you in terms of equality, constructor, and property code. It's just a shame we couldn't name them or add more functionality later. Record types fix all of this and more. Finally, I want to highlight a few features that involve a little more thinking outside the box.

15.6.3 Even more features in brief

Although some minor features are more likely to make it into C# 8, they're not as interesting as the ones I discuss here. Remember, you can always check GitHub to learn more about what might be included and its up-to-date status.

TYPE CLASSES (AKA CONCEPTS, SHAPES, OR STRUCTURAL GENERIC CONSTRAINTS)

Although generics are great for many situations, they have limitations. There are "shapes" of data types that can't be expressed with generics, such as operators and constructors. Although you can require that a type argument has a parameterless constructor, you can't require that it has a constructor with a specific parameter list. Additionally, at times types can have the same shape in some useful way but not implement any common interfaces or have any common base classes other than `System.Object`. *Type classes* would be a new kind of type to address these concerns. They'd be a little like interfaces, but the implementing class wouldn't need to know about them. You would be able to constrain a generic type parameter by the type class instead.

This has the potential to be powerful but somewhat confusing; I'm of two minds about it myself. It's likely to require runtime changes in order to execute efficiently. It may take C# developers (or me, at least) a while to work out when it's useful and when it's just confusing. Adding a whole new kind of type at this stage in the language's evolution feels like a giant step. For all these caveats, this feature definitely fills a gap: where you need this functionality, the current tools don't offer any clean solutions.

EXTENSION EVERYTHING

At the time of this writing, this has a milestone of X.0 in GitHub, but I wouldn't be overly surprised to see it move up the priority list. The name does a good job of

explaining the feature: the concept of extension methods would be applied to other member types, such as properties, constructors, and operators. It may also allow static extension members to be introduced—ones that look like they’re static methods on the extended type. (For example, you could write a method in `StringExtensions` that could be called as `string.IsNullOrEmptyTabs` as a more specific version of `string.IsNullOrEmptyWhiteSpace`.)

The syntax used for extension methods doesn’t lend itself to other member types, so it’s probable that a whole new syntax would be used instead. This might be an extension type that’s purely present to create multiple extension members all on one specific extended type.

Extension types still wouldn’t be able to introduce new state. Any extension properties would be likely to present a different view of existing properties. For example, you could have an extension property on `DateTime` called `FinancialQuarter` that knew your company’s financial reporting dates and used the existing `Year`/`Month`/`Day` properties to compute the appropriate quarter.

TARGET-TYPED NEW

Implicit typing with `var` can be useful for reducing clutter when long type names are involved. It doesn’t help for fields, though, because they can’t be implicitly typed. We still end up with code like this:

```
Dictionary<string, List<DateTime>> entryTimesByName =  
    new Dictionary<string, List<DateTime>>();
```

The *target-typed new* feature wouldn’t affect where you could use `var`. Instead, it would shorten the right-hand side of the declaration:

```
Dictionary<string, List<DateTime>> entryTimesByName = new();
```

Anytime the compiler can tell which type you probably mean when calling a constructor, you’d be able to leave out the type name entirely. This introduces interesting complexity with member invocations. For example, `Method(new())` would take the target type from the method parameter, which is fine until `Method` is generic or overloaded.

I love and hate this feature proposal, in roughly equal measure. It could certainly make code unreadable if used excessively, but almost any feature can be misused. On the other hand, I relish the possibility of removing the duplication of long field initialization.

I expect this to be even more controversial than default interface methods. We’ll see what happens, and you can be part of the conversation.

15.7 Getting involved

The C# design process is more open than ever before. Although a lot of work goes on in the background with Language Design Meetings (LDMs) in Microsoft offices, there’s plenty of room for community involvement, too. The GitHub repository at <https://github.com/dotnet/csharplang> is the place to start. It contains notes from

LDMs, proposals, discussions, and specifications. You're welcome to engage at any of the following levels:

- Trying out preview builds to see how well new features fit with your existing code
- Discussing currently proposed features
- Proposing new features
- Prototyping new features in Roslyn
- Helping draft language in the specification for new features
- Spotting mistakes in the existing specification (it happens!)

You may feel it's a better use of your time to wait for full releases with complete documentation and a polished implementation. That's perfectly fine, too. It's easy enough to dip your toe in the water at any time, if only to look at the set of proposed features for a given milestone.

This open design process is relatively new, and I expect it to be fine-tuned over time. I'd be surprised if the team ever went back to a more closed process. Although community engagement like this is expensive in terms of time, there are huge benefits in making sure the new features are ones developers really need.

Conclusion

There's been a lot more text than code in this chapter, mostly because I don't want to present too much code that'll be wrong by the time C# 8 ships. I doubt that all the features I've described will be present in C# 8, but I think it's at least likely that some of them will be. I'd be surprised if nullable reference types or the pattern-related features didn't make it into C# 8.

What comes beyond that? Well, minor releases in the C# 8 line, presumably, and then on to C# 9. Some of the features of C# 9 are probably already on GitHub as proposals, but I suspect there'll be some that haven't been talked about at all yet. I expect C# to continue to evolve to meet the needs of developers as the computing landscape changes.

appendix

Language features by version

This book is mostly organized by version, but it can be difficult to get a sense of the features introduced in each version at a glance. This is particularly true for the features introduced in minor versions of C# 7, which typically improve a feature introduced in C# 7.0.

Additionally, it can be useful to know whether a language feature requires runtime or framework support or whether it's pure compiler magic. This appendix aims to make all of this information available as simply as possible.

One aspect I haven't mentioned is how generic type inference has evolved over versions. It's changed many times and usually in ways that are too complicated to capture in just a few words. I suggest you take it as a given that anytime a new version is introduced, generic type inference may have improved.

Feature	Notes and requirements	Section
C# 2		
Generics	Runtime and framework support required.	2.1
Nullable value types	Runtime and framework support required.	2.2
Method group conversions		2.3.1
Anonymous methods		2.3.2
Delegate covariance and contravariance ^a		2.3.3
Iterators (<code>yield return</code>)		2.4
Partial types		2.5.1
Static classes		2.5.2

^a This refers to constructing a delegate from a method with a compatible but not identical signature. This isn't the same as generic variance introduced in C# 4.

Feature	Notes and requirements	Section
C# 2 (continued)		
Separate getter/setter access on properties		2.5.3
Namespace alias qualifier :: syntax		2.5.4
The global namespace alias		2.5.4
Extern aliases		2.5.5
Fixed-size buffers		2.5.6
<code>InternalsVisibleToAttribute</code> support	Runtime and framework support required.	2.5.7
C# 3		
Partial methods		2.5.1
Automatically implemented properties		3.1
Implicitly typed local variables (var)		3.2.2
Implicitly typed arrays (new [])		3.2.3
Object initializers		3.3.2
Collection initializers		3.3.3
Anonymous types		3.4
Lambda expressions (delegates)		3.5
Lambda expressions (expression trees)	Framework support required (expression tree types).	3.5.3
Extension methods	Framework support required (attribute).	3.6
Query expressions		3.7
C# 4		
Dynamic typing	Framework support required (called the <i>Dynamic Language Runtime</i> but not part of the runtime).	4.1
Optional parameters		4.2
Named arguments		4.2
Linked primary interop assemblies	Runtime and framework support required.	4.3.1
Special rules for optional parameters in COM		4.3.2
Access to named indexers (COM only)		4.3.3

Feature	Notes and requirements	Section
C# 4 (continued)		
Generic variance for interfaces and delegates	Framework changes to existing interfaces and delegates. (Runtime support was already present.)	4.4
Implementation change to lock statements	Framework support required: <code>Monitor.Enter(object, ref bool).</code>	Third edition, section 13.4.1
Implementation changes to field-like events		Third edition, section 13.4.2
Field-like event access within the declaring class		Third edition, section 13.4.2
C# 5		
Async/await	Framework support (task types and additional infrastructure used by the compiler).	Chapters 5 and 6
Changes to <code>foreach</code> iteration variable capture	Change in behavior, but only for code that was almost certainly broken in previous versions.	7.1
Caller information attributes	Framework support (the attributes themselves).	7.2
C# 6		
Read-only automatically implemented properties		8.2.1
Initializers for automatically implemented properties		8.2.2
Remove requirement to call <code>this()</code> in constructors for structs containing automatically implemented properties		8.2.3
Expression-bodied members		8.3
Interpolated string literals	Additional support for <code>FormattableString</code> when that class and <code>FormattableStringFactory</code> are available.	9.2, 9.3
The <code>nameof</code> operator		9.5
The <code>using static</code> directive		10.1
Object initializers using indexers		10.2.1
Collection initializers using extension Add methods		10.2.2
The null conditional <code>?.</code> operator		10.3

Feature	Notes and requirements	Section
C# 6 (continued)		
Exception filters		10.4
Removed restrictions on awaiting in <code>try/catch</code> , <code>try/finally</code> , and <code>try/catch</code> statements		5.4.2
C# 7.0		
Tuples	Framework support (ValueTuple types).	11.2–11.4
Deconstruction via <code>Deconstruct</code> methods	Required ValueTuple types to be present until C# 7.2 compiler, but not a C# 7.2 language feature. (Implementation change, effectively.)	12.1, 12.2
Initial patterns: constant patterns, type patterns, <code>var</code> patterns		12.4
Use of patterns with the <code>is</code> operator		12.5
Use of patterns in switch statements, including guard clauses (<code>when</code>)		12.6
Ref locals		13.2.1
Ref return		13.2.2
Binary integer literals		14.3.1
Underscore separators in numeric literals		14.3.2
Returning custom task types from <code>async</code> methods	Framework support required (attributes).	5.8
More kinds of expression-bodied members		8.3.3
C# 7.1		
The <code>default</code> literal		14.5
Improvements to type patterns matching against generic values		12.4.2
Async entry points (<code>async Task Main</code>)		5.9
Inferred tuple element names		11.2.2
C# 7.2		
Allow the conditional <code>? :</code> operator to work with <code>ref</code>		13.2.3

Feature	Notes and requirements	Section
C# 7.2 (continued)		
ref readonly locals and return types	Methods returning <code>ref readonly</code> can be called only by compilers that understand them. Additionally, <code>InAttribute</code> is required at compile time but has been present since .NET 1.1 and .NET Standard 1.1.	13.2.4
in parameters	Requires <code>IsReadOnlyAttribute</code> , but that's bundled in the output if it's missing from the target framework.	13.3
Read-only structs	Requires <code>IsReadOnlyAttribute</code> as noted the preceding entry.	13.4
Extension methods with <code>ref/in</code> parameters		13.5
Ref-like structs	Requires <code>IsReadOnlyAttribute</code> as noted previously. Additionally, ref-like structs have <code>ObsoleteAttribute</code> applied to them with a specific message. Ref-like-struct-aware compiler versions ignore this, but earlier compilers will prevent the type being used.	13.6
<code>stackalloc</code> support for <code>Span<T></code>	Framework support required.	13.6.2
Nontrailing named arguments		14.6
The <code>private protected</code> access modifier		14.7
Underscore separators in numeric literals directly after the <code>0x</code> or <code>0b</code> base specifier		14.3.2
C# 7.3		
Access to fixed-sized buffers via fields without <code>fixed</code> statements		2.5.6
<code>==</code> and <code>!=</code> operators for tuples	Availability of tuples, but no new requirements.	11.3.6
Use of <code>pattern</code> and <code>out</code> variables in field, property, and constructor initializers		14.2.2
Reassignment of <code>ref</code> locals		13.2.1
Initializers in <code>stackalloc</code> statements		13.6.2
Pattern-based <code>fixed</code> statements using <code>GetPinnableReference</code>		13.6.2

Feature	Notes and requirements	Section
C# 7.3 (continued)		
Generic type constraints now permitted on <code>Enum</code> and <code>Delegate</code>		14.8.1
New generic type constraint of <code>unmanaged</code>	Types and methods with the <code>unmanaged</code> constraint can be used only by compilers recent enough to understand it. Also required <code>UnmanagedType</code> enum, available since .NET 1.1 and .NET Standard 1.1.	14.8.1
Placement of attributes of fields backing automatically implemented properties		14.8.3

index

Symbols

! operator 46, 445–447
!= operator 337, 477
& operator 45–46
&& operator 373
== operator 301, 337, 368, 477
=> syntax 8, 92, 243, 454
. operator 49, 475
?: operator 86, 273, 392, 431, 476
?? operator 48–49, 302
^ operator 46
_ discards 357
| operator 45–46

A

Action delegate 49
add signatures
 general-purpose 296
 specialized 296–297
Add() method 27, 55, 85, 245, 292, 298, 475
Add(key, value) method 85
AddOrUpdate method 293
AddRange method 296
Advanced Build Settings dialog box 13
AggregateException 174–175
aliases 281
 extern 71–72
 global namespace 71

AND operator 46
anonymous classes, Java 87
anonymous functions
 asynchronous 180–181
 overview of 129–130
anonymous methods 6, 50–52, 91
anonymous object creation expression 87
anonymous types 5, 86–91, 130–131
 as alternatives to tuples 347–348
 behavior of 86–89
 compiler-generated types 89–90
 limitations of 90–91
 syntax of 86–89
Any() method 128
application programming interfaces (APIs)
 nonpublic 348–349
 null-returning 304
args parameter array 256
ArgumentException 342
ArgumentNullException 106, 249, 448–449
ArgumentOutOfRangeException 222
arguments
 iterator/async 426–427
 named 433–435
 validating 177–179, 277
 See also type arguments
arity 28–29
array types 282
ArrayList 23, 25
arrays 22, 79–81
as operator 48, 118, 371
as-followed-by-if statements 368
AsQueryable() method 130
assemblies, interop 139–140
Assert.ThrowsAsync method 191
assignment operator 102

assignments
 deconstruction
 to existing properties 357–361
 to existing variables 357–361
 definite 419

AsTask method 182

async methods 10

async/await feature 10–11, 151

asynchronous code
 asynchronous anonymous functions 180–181
 asynchronous method declarations 160–162
 parameters in asynchronous methods 162
 return types from asynchronous
 methods 161–162

asynchronous method flow 168–180
 awaitable pattern members 173–174
 awaited 168–169
 evaluating await expressions 169–172
 exception unwrapping 174–176
 method completion 176–180

avoid mixing with synchronous code 190

await expressions 162–166
 awaitable patterns 163–165
 restrictions on 165–166

control flow and MoveNext() 210–216
 awaiting within loops 212–213
 awaiting within try/finally blocks 213–216
 control flow between await expressions 211

custom task types 150–182, 186–219

execution contexts 216–218

flow 216–218

functions 152–155

implementing MoveNext() method 205–210
 examples of 205–206
 general structure of 207–209
 zooming into await expressions 209–210

main methods in C# 7.1 186–187

structure of generated code 195–205
 MoveNext() method 202–204
 SetStateMachine method 204–205
 state machine boxing 204–205
 structure of state machines 199–202
 stub method 198–199

usage tips 187–192
 allowing cancellation 190–191
 avoiding context capture using
 ConfigureAwait 187–189
 enabling parallelism 189–190
 testing asynchrony 191–192

wrapping return values 166–168

asynchronous integration 461–466

asynchronous iterators 465–466

asynchronous resource disposal with using
 await 461–462

with foreach await 462–465

asynchronous iterations
 with foreach await 462–465
 overview of 465–466

asynchronous methods
 flow 168–180
 await 168–169
 awaitable pattern members 173–174
 evaluating await expressions 169–172
 exception unwrapping 174–176
 method completion 176–180
 method declarations 160–162
 parameters in asynchronous methods 162
 return types from asynchronous
 methods 161–162

modeling 158–160

parameters in 162
 return types from 161–162

asynchronous resource disposal 461–462

asynchrony 10–11
 overview of 155–160
 fundamentals of asynchronous
 execution 155–157
 modeling asynchronous methods 158–160
 synchronization contexts 157–158
 testing 191–192

AsyncTaskMethodBuilder class 198, 200, 204, 217–218

AsyncTaskMethodBuilderAttribute 200

AsyncVoidMethodBuilder 200

attributes 278–279
 expressing nullable intent with 447–448
 for fields backing automatically implemented
 properties 437–438
See also caller information attributes

automatically implemented properties 8, 76, 238–242
 attributes for fields backing 437–438
 initializing 239–240
 read-only 238–239
 in structs 240–242

await expressions 10, 162–166
 awaitable patterns 163–165
 control flow between 211
 evaluating 169–172
 foreach await 462–465
 overview of 168–169
 restrictions on 165–166
 using await statement 461–462

await operator 152

awaitable patterns 163–165, 173–174

awaiters 201, 204, 207–208

awaiting
 within loops 212–213
 within try/finally blocks 213–216

AwaitOnCompleted 210, 217

AwaitUnsafeOnCompleted method 186, 210, 217–218

B

bang operators
 best practices 448–449
 overview of 445–447
BenchmarkDotNet project 399
BigQuery 434
binary compatible 396
binary integer literals 429–430
binary-ish operator 459
BindableObject class 226
Binder class 127
binding 77, 115
 See also dynamic binding
Blazor 14
boilerplate code 6
Boolean comparisons 301–302
boxing
 behavior 42–43
 state machines 204–205
buffers. *See* fixed-size buffers
BuilderType property 185
Button classes 70
Button.OnClick method 155
byte-wise access 183
ByteStream 183

C

C# 2 21–74
 generics 22–38
 advantages of 25–29
 default operators 34–37
 generic type initialization 37–38
 generic type state 37–38
 identifying 29–30
 type constraints 32–34
 type inference for type arguments to
 methods 30–32
 typeof operators 34–37
 iterators 53–66
 evaluating finally blocks 58–62
 evaluating yield statements 56–57
 implementation sketches 62–66
 lazy execution 55–58
 overview of 54–55
 language features in 473–474
 minor features of 66–74
 fixed-size buffers 73
 InternalsVisibleTo 73–74
 namespace aliases 70–72

partial types 67–69
pragma directives 72
separate getter/setter access for
 properties 69–70
 static classes 69
nullable value types 38–49
 expressing absence of information 39–40
 language support 43–49
 Nullable struct 40–43
 simplified delegate creation 49–53
 anonymous methods 50–52
 delegate compatibility 52–53
 method group conversions 50
C# 3 75–112
 anonymous types 86–91
 behavior of 86–89
 compiler-generated types 89–90
 limitations of 90–91
 syntax of 86–89
 automatically implemented properties 76
 collection initializers 81–86
 benefits of single expressions for
 initialization 86
 overview of 81–83
 extension methods 103–107
 chaining method calls 106–107
 declaring 103–104
 invoking 104–106
 implicit typing 77–81
 implicitly typed arrays 79–81
 implicitly typed local variables 78–79
 typing terminology 77
 Lambda expressions 91–103
 capturing variables 94–101
 expression trees 101–103
 syntax of 92–94
 language features in 474
 LINQ 111–112
 object initializers 81–86
 benefits of single expressions for
 initialization 86
 overview of 81–83
 query expressions 107–110
 LINQ syntax 110
 range variables 108–109
 translating from C# to C# 108
 transparent identifiers 108–109
C# 4 113–149
 COM interoperability improvements 138–143
 linking primary interop assemblies 139–140
 named indexers 142–143
 optional parameters in COM 140–142
 dynamic typing 114–133
 advantages of 127–131
 anonymous functions 129–130

- C# 4 (*continued*)
 - anonymous types 130–131
 - dynamic behavior 119–124
 - explicit interface implementation 131
 - extension methods 128–129
 - generating IL for 125–127
 - generics and 127–128
 - limitations of 127–131
 - overview of 114–119, 124–127
 - usage suggestions 131–133
 generic variance 143–149
 - examples of 143–144
 - in practice 147–149
 - restrictions on using variance 145–147
 - syntax for variance in interface declarations 144–145
 language features in 474–475
 named arguments 133–138
 - determining meaning of method calls 135–137
 - impact on versioning 137–138
 optional parameters 133–138
 - determining meaning of method calls 135–137
 - impact on versioning 137–138
 parameters with default values 134–135
- C# 5 220–232
 - caller information attributes 222–232
 - behavior of 222–223
 - corner cases of 226–232
 - logging 224
 - with old versions of .NET 232
 - simplifying `IPropertyChanged` implementations 224–226
 - capturing variables in foreach loops 220–222
 - language features in 475
- C# 6
 - language features in 475–476
 - restrictions on expression-bodied members in 247–249
- C# 7 415–438
 - custom task types in 182–186
 - building 184–186
 - `ValueTask` 182–184
 - improvements to numeric literals 429–431
 - binary integer literals 429–430
 - underscore separators 430–431
 - local methods 415–427
 - implementing 420–425
 - usage guidelines 425–427
 - variables within 417–420
 - out variables 427–429
 - inline variable declarations for out parameters 427–428
 - restrictions lifted in C# 7.3 for out variables 428–429
- restrictions lifted in C# 7.3 for pattern variables 428–429
- throw expressions 431–432
- C# 7.0
 - language features in 476
 - patterns in 367–372
 - constant patterns 367–368
 - type patterns 368–371
 - var pattern 371–372
- C# 7.1
 - asynchronous code in 186–187
 - default literals in 432–433
 - inferred element names for tuple literals in 323–324
 - language features in 476
- C# 7.2
 - accessing private protected in 435
 - conditional ?: operator in 392–393
 - declaring structs as readonly in 401–405
 - implicit copying with read-only variables 401–403
 - readonly modifier for structs 403–404
 - XML serialization implicitly read-write 404–405
 - in parameters in 395–400
 - compatibility 396–397
 - extension methods with 405–408
 - guidance for 399–400
 - mutability of 397–398
 - overloading with 398–399
 - language features in 476–477
 - nontrailing named arguments in 433–435
 - ref parameters in 405–408
 - ref readonly in 393–395
 - ref values in 392–393
 - ref-like structs in 408–414
 - IL representation of 414
 - rules for 409–410
 - Span 410–414
 - stackalloc 410–414
- C# 7.3
 - equality operators in 337–338
 - improved access to fixed-size buffers in 73
 - improvements in 435–438
 - attributes for fields backing automatically implemented properties 437–438
 - generic type constraints 435–436
 - overload resolution 436–437
 - inequality operators in 337–338
 - language features in 477–478
 - pattern-based fixed statements in 413–414
 - restrictions lifted for out variables 428–429
 - restrictions lifted for pattern variables 428–429
 - stackalloc with initializers in 413

- C# 8 439–471
asynchronous integration 461–466
asynchronous iteration with foreach
 await 462–465
asynchronous iterators 465–466
asynchronous resource disposal with using
 await 461–462
default interface methods 466–468
extensions 469–470
indexes 458–460
 applying 459–460
 index literals 458–459
 index types 458–459
nullable reference types 440–453
 advantages of 440–441
 bang operator 445–447
 changing meaning when using reference
 types 441–442
 at compile time 443–445
 entering 442–443
 at execution time 443–445
 future improvements to 449–453
 migration of 447–449
ranges 458–460
 applying 459–460
 range literals 458–459
 range types 458–459
record types 468–469
recursive pattern matching 455–458
 deconstruction patterns 456–457
 matching properties in patterns 455–456
 omitting types from patterns 457–458
switch expressions 453–455
target-typed new 470
type classes 469
Caliburn Micro MVVM framework 226
call sites 127
callback 156
caller information attributes 8, 222–232
 attributes with 230–232
 behavior of 229–223
 corner cases of 226–232
 dynamically invoked members 226–227
 implicit constructor invocations 228–229
 non-obvious member names 228
 query expression invocations 229–230
 logging 224
 with old versions of .NET 232
 simplifying INotifyPropertyChanged
 implementations 224–226
CallerFilePathAttribute 222
CallerLineNumberAttribute 222
CallerMemberNameAttribute 222–224, 226, 277
cancellations
 allowing 190–191
 handling 179–180
CancellationToken 135, 156, 174, 179, 432
CancellationTokenSource 174, 179
candidate types 80
captured variables 221, 417–418
capturing
 ref parameters of enclosing methods 418
 variables 94–101
 in foreach loops 220–222
implementing captured variables with generated class 95–97
multiple instantiations of local variables 97–99
from multiple scopes 99–101
carriage-return line-feed separators 260
case labels 376–377
case-specific exception filters 313–314
catch block 305, 307
catch clause 307
ceremony 6
chaining method calls 106–107
checking nullability 452–453
class keyword 33
classes
 generated 95–97
 static 69
 type 469
clauses, guard 375–376
closed constructed types 36–37
closures 7, 51
CLR. *See* Common Runtime Language
CodedInputStream class 184
collection initializers 7, 81–86
 benefits of single expressions for
 initialization 86
 enhancements to 290–299
 extension methods in 294–298
 creating general-purpose add signatures 296
 creating specialized add signatures 296–297
 reexposing existing methods 297–298
 overview of 81–83
 test code vs. production code 298–299
colons 70, 273
COM. *See* Component Object Model
Common Language Runtime (CLR), tuples
 in 338–346
 element name handling 339–341
 extension methods 346
 implementing tuple conversions 341
 nongeneric ValueTuple struct 346
 regular equality and ordering
 comparisons 342–343
 string representations of 341–342
 structural equality and ordering
 comparisons 343–345
System.ValueTuple 338–339
wombles 345

- community 14–15
 Community Technology Preview (CTP) 193
 Company class 39
 CompareTo method 34
 comparisons
 Boolean, handling 301–302
 ordering
 regular equality and 342–343
 structural equality and 343–345
 compatibility
 of delegates 52–53
 of in parameters 396–397
 compile time 443–445
 Compile() method 103
 compile-time error 419
 compiler handling
 of Deconstruct calls 364–365
 of interpolated string literals 261
 compiler-generated types 89–90
 compilers
 overview of 117–118
 providing with semantic information 450–451
 compiling expression trees 102–103
 Complete state 195
 completed tasks 170
 Component Object Model (COM)
 improvements to interoperability 138–143
 linking primary interop assemblies 139–140
 named indexers 142–143
 optional parameters in 140–142
 composite format string 253
 computed properties
 property change notifications for 277–278
 read-only 242–245
 delegating properties 244–245
 pass-through properties 244–245
 performing simple logic on another piece of state 245
 conditional ?: operator 392–393
 conditional operators. *See* null conditional operators
 ConfigureAwait method 184, 187–189
 ConfigureAwaitChecker.Analyzer NuGet package 189
 Console.WriteLine method 51, 212, 259, 442
 constant patterns 367–368
 constraints, type
 generic 435–436
 overview of 32–34
 construction 6–8
 constructor constraint 33
 constructor invocations 228–229
 context capture 187–189
 contexts
 execution 216–218
 synchronization 157–158
 continuations 155–156
 contravariance 144
 control flow
 between await expressions 211
 MoveNext() and 210–216
 awaiting within loops 212–213
 awaiting within try/finally blocks 213–216
 Control.BeginInvoke 157
 Control.Invoke 157
 conversion constraint 33
 conversions 44–45
 between tuple types 334–336
 generic variance conversions 336
 tuple type identity conversions 335–336
 explicit 331–332
 to expression trees 102
 generic variance 336
 identity conversions of tuple types 335–336
 implicit 330–331
 method group conversions 50
 of tuples 329–338
 element name checking in inheritance 336–337
 equality operators in C# 7.3 337–338
 implementing 341
 inequality operators in C# 7.3 337–338
 uses of 336
 to tuples types 330–334
 copying, implicitly 401–403
 corner cases, of caller information attributes 226–232
 attributes with caller information
 attributes 230–232
 dynamically invoked members 226–227
 implicit constructor invocations 228–229
 non-obvious member names 228
 query expression invocations 229–230
 Count() method 8, 172, 467
 covariance 144
 cruft 6
 CTP. *See* Community Technology Preview
 CultureInfo 255
 CultureInfo.CurrentCulture property 257
 CultureInfo.InvariantCulture property 257, 263
 cultures
 default 257
 formatting FormattableString in 263–265
 noninvariant 265
 Current property 56, 64

D

-
- damn it operators. *See* bang operators
 data access with LINQ 9–10
 database access 119–120

DateTime 363
DateTime.Add(TimeSpan) method 85
DateTimeOffset 104
debug 194
declarations
 delegate declarations 144–145
 interface declarations 144–145
 variable declarations 427–428
 See also asynchronous methods, method declarations
declaring
 extension methods 103–104
 local methods after declaring captured variables 417–418
decompilers 196, 213
Deconstruct methods 364–365, 379, 476
deconstruction 353–380
 assignments to existing properties 357–361
 assignments to existing variables 357–361
 to new variables 355–357
 of nontuple types 361–365
 compiler handling of Deconstruct calls 364–365
 extension deconstruction methods 363–364
 instance deconstruction methods 362
 overloading 363–364
opportunities for 379
of tuple literals 361
 of tuples 354–361
deconstruction patterns 456–457
default cultures 257
default indexer 142
default interface methods 466–468
default literals 135, 432–433
default operators 34–37, 432
default values 134
 changing 137–138
 parameters with 134–135
definite assignment 240, 419
delegate creation expressions 50
delegate declarations 144–145
delegate keyword 92
delegates 74
 compatibility of 52–53
 compiling expression trees to 102–103
 creating 49–53
 anonymous methods 50–52
 method group conversions 50
delegating properties 244–245
dereferencing properties 299–300
Dictionary collection 85
directives. *See* pragma directives
Dispose() method 55, 61, 66
Distinct() method 343
DLR. *See* Dynamic Language Runtime

dot operator 163
dot syntax 110
DownloadString 153
dynamic behavior 119–124
 dynamic view of Json.NET 121
 examples of database access 119–120
 ExpandoObject 120–121
 implementing 121–124
dynamic binder
 element names 351–352
 high element numbers 352
 tuples and 351–352
dynamic binding
 dynamic values and 118
 overview of 116–117
dynamic formatting 272
Dynamic Language Runtime (DLR) 125
dynamic typing 77, 114–133
 anonymous functions 129–130
 anonymous types 130–131
 applying dynamic binding 116–117
 compilers 117–118
 dynamic behavior 119–124
 dynamic view of Json.NET 121
 examples of database access 119–120
 ExpandoObject 120–121
 implementing 121–124
 dynamic values and dynamic binding 118
 dynamic values and static types 118–119
 extension methods 128–129
 generating IL for 125–127
 generics and 127–128
 implementing explicit interfaces 131
 libraries for 133
 limitations of 127–131
 overview of 114–119, 124–127
 usage suggestions 131–133
 common members without common interface 133
 reflection 132
dynamic values
 dynamic binding and 118
 static types and 118–119
dynamically invoked members 226–227
DynamicMetaObject 122–123
DynamicObject class 133, 149

E

element initializers 84
element names
 checking in inheritance 336–337
 dynamic binder and 351–352
 at execution time 340–341
 handling 339–341

- element names (*continued*)
 inferred 323–324
 in metadata 340
 in tuple literal conversions 332–334
- elements
 accessing by name 325–326
 accessing by position 325–326
 copying 27
- encapsulation 91
- enclosing methods 418
- end users 271–272
- entry points 187
- Enumerable class 111
- Enumerable.Count() method 289, 467
- enums 29
- equality 91
 regular 342–343
 structural 343–345
- equality operators 45, 337–338
- Equals() method 41, 89, 302, 337, 449
- evaluation order 377–379
- event subscription 162
- EventHandler 49–51, 303
- EventInfo 282
- events, raising 303
- exception filters 305–315
 case-specific 313–314
 logging as side effect 312–313
 rethrowing 314–315
 retrying operations 311–312
 syntax of 306–310
 catching exception type multiple times 310
 two-pass exception model 307–310
- exception types 310
- exceptions
 lazy 177–179
 unwrapping 174–176
- exclusive OR operator 46
- Executing state 195
- execution
 asynchronous 155–157
 element names at time of 340–341
- execution contexts 216–218
- execution time 443–445
- ExecutionContext class 216
- ExecutionContext.Capture 217
- ExecutionContext.Run 217
- ExpandoObject 120–121
- explicit conversions 331–332
- explicit interfaces 131
- explicit typing 77
- exposing methods 297–298
- expression bodies 92
- Expression class 102, 124
- expression trees 10, 92, 101–103
- compiling to delegates 102–103
- limitations of conversions to 102
- expression-bodied indexers 245–247
- expression-bodied lambda expressions 102
- expression-bodied members 8, 242–251
 guidelines for 249–251
- read-only computed properties 242–245
 delegating properties 244–245
 pass-through properties 244–245
 performing simple logic on another piece of state 245
- restrictions on 247–249
- expression-bodied methods 245–247
- expression-bodied operators 245–247
- expressions
 reevaluating 272–273
 single 86
 switch 453–455
 throw expressions 431–432
 See also await expressions
- extended type 104
- [Extension] attribute 104
- extension deconstruction methods 363–364
- extension methods 103–107
 chaining method calls 106–107
 in collection initializers 294–298
 creating general-purpose add signatures 296
 creating specialized add signatures 296–297
 reexposing existing methods 297–298
- declaring 103–104
- with in parameters in C# 7.2 405–408
- invoking 104–106
- overview of 128–129
- ref extension methods 407–408
- with ref parameters in C# 7.2 405–408
- ref/in parameters in 405–407
- static directives and 288–290
- extensions 469–470
- extern aliases 71–72

F

-
- false operator 45
- fast path 207
- fetch result 171
- Fibonacci sequence 327, 375
- fields 350–351
 attributes for 437–438
 read-only 419–420
 ref fields 389
- File.ReadLines method 61
- filters. *See also* exception filters
- finally blocks 58–62, 307, 310, 314
- FirstOrDefault 304
- fixed statements, pattern-based 413–414

fixed-size buffers 73
 flow 216–218
See also asynchronous methods, flow
 fluent syntax 110
 for statement 249
 foreach await statement 462–465
 foreach loops 220–222
 Format method 253
 format string 254
FormattableString class 330, 475
 formatting in noninvariant culture 265
 formatting in specific cultures 263–265
 localization using 261–270
 with older versions of .NET 268–270
 uses for 265–268
FormattableStringFactory class 268, 475
 formatting
 with default cultures 257
 deferring for strings 273–274
 dynamically 272
FormattableString in noninvariant cultures 265
FormattableString in specific cultures 263–265
 for machines 257–258
 for readability 274–275
 strings in .NET 253–258
 custom formatting with format strings 253–255
 localization 255–258
 simple string formatting 253
 strings in interpolated string literals 259
 from clause 108–109
 function member 17
 functions. *See* anonymous functions

G

generated classes 95–97
 generated code, structure of 195–205
`MoveNext()` method 202–204
`SetStateMachine` method 204–205
 state machine boxing 204–205
 structure of state machines 199–202
 stub method 198–199
 generators 54
 generic methods 28–29
 generic type constraints 435–436
 generic types 35
 arity of 28–29
 initializing 37–38
 states 37–38
 generic variance 53, 143–149
 conversions 336
 examples of 143–144
 in practice 147–149
 restrictions on using variance 145–147
 syntax for variance in delegate declarations 144–145

syntax for variance in interface declarations 144–145
GenericParameter 378
 generics 4, 22–38
 advantages of 25–29
 arity of generic methods 28–29
 arity of generic types 28–29
 type arguments 26–28
 type parameters 26–28
 default operators 34–37
 dynamic typing and 127–128
 generic type initialization 37–38
 generic type state 37–38
 identifying 29–30
 null-inconsistent generics 449
 overview of 281, 451
 type constraints 32–34
 type inference for type arguments to methods 30–32
`typeof` operators 34–37
`GetAsyncEnumerator()` method 463, 465
`GetAwaiter()` method 163, 165, 187, 201, 209, 361
`GetEnumerator()` method 8, 56, 64
`GetHashCode()` method 41, 89, 449
`GetPinnableReference` method 414, 477
`GetResult()` method 164–165, 174, 176, 187, 209
`GetStringAsync()` method 153, 175
 getter/setter access 69–70
`GetType()` method 35, 43, 340
`GetValueOrDefault()` method 41
 global namespace aliases 71
 globalization 255
 Google.Protobuf package 184
 goto statement 206, 212–213, 378
 guard clauses 371, 375–376
 Guid 135
`Guid.NewGuid` 410

H

handling catch block 307
 hash codes 91
`HasValue` property 40–42, 44
`Height` property 278
 HotSpot JIT compiler, Java 11
`HttpClient` 83, 151
`HttpClient.GetStringAsync` 178
`HttpRequestException` 175

I

`IAsyncEnumerator` interface 463, 465
`IAsyncStateMachine` interface 199–200
`IAsyncStateMachine.SetStateMachine` 218

IComparable interface 342–343
 IComparable<T> interface 34
 ICriticalNotifyCompletion method 216–217, 219
 ICriticalNotifyCompletion.UnsafeOnCompleted
 217–218
 identifiers
 accessing with nameof 275–283
 common uses of nameof 277–279
 nameof examples 275–276
 tricks when using nameof 280–283
 transparent 108–109
 identity conversions 53, 146, 335–336
 IDisposable interface 53, 61, 163, 461
 IDynamicMetaObjectProvider interface 120, 122,
 133, 149
 IEnumerable interface 28
 IEnumerable.GetEnumerator() method 55
 IEnumerator interface 61
 IEquatable interface 343
 IFormatProvider interface 255
 IFormattable interface 32–33, 259, 262
 IFormattable.ToString(string, IFormatProvider)
 method 33
 IIS. *See* Internet Information Server
 IL. *See* Intermediate Language
 ILogger interface 224
 implicit constructor invocations 228–229
 implicit conversions 77, 330–332
 implicit copying 401–403
 implicit typing 6, 77–81
 implicitly typed arrays 79–81
 implicitly typed local variables 5, 78–79
 importing static members 285–288
 in extension methods 407–408
 in parameters, in C# 7.2 395–400
 compatibility 396–397
 extension methods with 405–408
 guidance for 399–400
 mutability of 397–398
 overloading with 398–399
 InAttribute attribute 395–396, 477
 index literals 458–459
 index types 458–459
 IndexerNameAttribute 228
 indexers
 expression-bodied 245–247
 named 142–143
 null conditional operators and 302–303
 in object initializers 291–294
 indexes 458–460
 applying 459–460
 index literals 458–459
 index types 458–459
 inequality operators 337–338
 inference. *See* type inference
 inferring element names 323–324
 infoof operator 282
 information, absence of 39–40
 inheritance 336–337
 initializers 413
 initializing 6–8
 automatically implemented properties 239–240
 benefits of single expressions for 86
 generic types 37–38
 ref locals 388–389
 initializing. *See also* collection initializers
 inline variable declarations 427–428
 INotifyCompletion interface 216, 219
 INotifyCompletion.OnCompleted method 217
 INotifyPropertyChanged 224–226, 277
 instance deconstruction methods 362
 instance method 94
 instantiations of local variables 97–99
 int.TryParse 427–428
 integration. *See* asynchronous integration
 interface declarations 144–145
 interface methods 466–468
 interfaces
 common members without common
 interface 133
 explicit 131
 interior pointer 387
 Intermediate Language (IL)
 generating for dynamic types 125–127
 representation of ref-like structs 414
 InternalsVisibleToAttribute attribute 67, 73–74,
 91, 131, 397, 474
 Internet Information Server (IIS) 13
 interop assemblies 139–140
 interpolated string literals 9
 compiler handling of 261–263
 formatting strings in 259
 interpolated verbatim string literals 259–260
 limitations of 272–273
 overview of 258–261
 simple interpolations 258–259
 interpolations 258–259
 InvalidCastException 24
 InvalidOperationException 42, 455
 invariance 144
 invariant culture 257
 Invariant method 263–264, 270
 invocations
 implicit constructor invocations 228–229
 query expression invocations 229–230
 Invoke method 303
 invoking extension methods 104–106
 IOException 313
 is operator 48, 118, 370, 372–374, 476
 IsFixedSize property 131

IsReadOnlyAttribute 396, 477
[IsRefLikeAttribute] attribute 414
IStructuralComparable interface 343
IStructuralEquatable interface 343
Items property 83
iteration variables 220
iterations
 asynchronous 462–465
 overview of 448
iterator blocks 18, 54
iterator/async arguments 426–427
iterators 53–66, 74
 asynchronous 465–466
 evaluation of finally blocks 58–62
 evaluation of yield statements 56–57
 implementing 62–66
 lazy execution 55–58
 overview of 54–55
IXmlSerializable 404–405

J

Java HotSpot JIT compiler 11
JIT (just-in-time) compilers 11
Json.NET 121

K

KeyNotFoundException 9
KeyValuePair 354

L

Lambda expressions 7, 18, 91–103, 244, 246
 capturing variables 94–101
 implementing captured variables with generated class 95–97
 multiple instantiations of local variables 97–99
 from multiple scopes 99–101
 expression trees 101–103
 compiling to delegates 102–103
 limitations of conversions to 102
 syntax of 92–94
Language Design Meetings (LDMs) 470
language support 43–49
 ? type suffix 43
 conversions 44–45
 lifted operators 45–46
 null literal 44
 nullable logic 46–48
 null-coalescing ?? operator 48–49
lazy exceptions 56, 177–179
lazy execution
 importance of 57–58
 overview of 55–56

LDMs. *See* Language Design Meetings
Length property 93, 109, 360–361
let clause 109
libraries for dynamic typing 133
lifted operators 45–46
lifting 44
linking 139
LINQ framework
 data access with 9–10
 overview of 111–112
 syntax of 110
LINQ queries 5
List collection 25–27, 36, 131
literals
 binary integer 429–430
 default 432–433
 index 458–459
 null 44
 range 458–459
 See also interpolated string literals
literal-to-type conversion 334
local methods 415–427
 declaring 417–418
 escaping containing code 423–425
 implementing 420–425
 interacting with definite assignment 419
 optimizing 426–427
 read-only fields and 419–420
 usage guidelines 425–427
 iterator/async argument validation 426–427
 suggestions for readability 427
variables within 417–420
 capturing ref parameters of enclosing methods 418
 capturing variables in scope 417
local variables
 implicitly typed 78–79
 multiple instantiations of 97–99
 overview of 349–350, 389
localization 255–258
 formatting for machines 257–258
 formatting with default cultures 257
 using FormattableString 261–270
 compiler handling of interpolated string literals 262–263
 formatting FormattableString in specific cultures 263–265
 uses for FormattableString 265–268
 using FormattableString with older versions of .NET 268–270
localized data 90
lock statement 310
logging
 caller information attributes 224
 as side effect 312–313

logic
 nullable 46–48
 performing on another piece of state 245
 loops
 awaiting within 212–213
 foreach 220–222

M

machine.builder field 199
 machine.builder.Start() method 199
 machine-readable strings 270–271
 Main() method 30, 59, 71, 126, 171, 307, 312, 387, 397, 416
 matching. *See* pattern matching
 matching properties in patterns 455–456
 MaxBy method 349
 member initializers 83
 members
 common members without common interface 133
 dynamically invoked 226–227
 non-obvious member names 228
 of other types, referring to 280
 static 285–288
 See also expression-bodied members
 message pump 172
 MessageBox.Show method 138
 metadata 340
 method calls
 chaining 106–107
 determining meaning of 135–137
 method completion 176–180
 argument validation 177–179
 handling cancellations 179–180
 lazy exceptions 177–179
 returning successfully 177
 method declarations 8
 See also asynchronous methods, method declarations
 method flow. *See* asynchronous methods, flow
 method group conversions 6, 50
 method returns 170
 method syntax 110
 [MethodImpl] attribute 416
 MethodInfo 132, 282
 methods
 anonymous 50–52
 enclosing 418
 expression-bodied 245–247
 generic 28–29
 interface 466–468
 partial 68–69
 reexposing 297–298
 type arguments to 30–32

See also asynchronous methods; extension methods; MoveNext() method
 Microsoft C# compiler 72
 Microsoft.Bcl package, NuGet 232
 Microsoft.CSharp.RuntimeBinder 127
 migration of nullable reference types 447–449
 best practices for using bang operators 448–449
 end results 449
 expressing nullable intent with attributes 447–448
 iterations 448
 null-inconsistent generics 449
 minor versions 12–13
 model-view-viewmodel (MVVM) 226
 modreq modifier 396
 Monitor.Exit method 166
 Monitor.TryEnter method 166
 MoreLinq package 349
 MoveNext() method 56–57, 59–63, 65–66, 178, 202–204, 467
 control flow and 210–216
 awaiting within loops 212–213
 awaiting within try/finally blocks 213–216
 control flow between await expressions 211
 examples of 205–206
 general structure of 207–209
 implementing 209–210
 MulticastDelegate 436
 mutability of in parameters 397–398
 mutable value types 198
 MVVM. *See* model-view-viewmodel

N

name attribute 304
 named arguments, nontrailing 433–435
 named indexers 142–143
 named types 348
 nameof operator 9, 225, 452, 475
 accessing identifiers with 275–283
 common uses of 277–279
 argument validation 277
 attributes 278–279
 property change notifications for computed properties 277–278
 examples of 275–276
 tricks when using 280–283
 array types 282
 generics 281
 namespaces 282–283
 nullable value types 282
 predefined aliases 282
 referring to members of other types 280
 simple name 282
 using aliases 281

- names
 - accessing elements by 325–326
 - of members 228
 - of parameters 137
 - simple name 282
 - See also* element names
- namespace aliases 70–72
 - extern aliases 71–72
 - global namespace 71
 - qualifier syntax 70–71
- namespaces 282–283
- nested patterns 456
- .NET platform
 - caller information attributes with 232
 - FormattableString with 268–270
 - formatting strings in 253–258
 - custom formatting with format strings 253–255
 - localization 255–258
 - simple string formatting 253
 - Noda Time 16–17, 242, 244, 289, 376
 - nonasync method 178
 - noncompleted tasks 170
 - nongeneric ValueTuple struct 346
 - noninvariant cultures 265
 - non-obvious member names 228
 - nonpublic APIs 348–349
 - nontrailing named arguments 433–435
 - nontuple types, deconstruction of 361–365
 - compiler handling of Deconstruct calls 364–365
 - extension deconstruction methods 363–364
 - instance deconstruction methods 362
 - overloading 363–364
 - Not started state 195
 - NotifyOfPropertyChanged method 226
 - NotImplementedException 85
 - NotSupportedException 467
 - NuGet (NuPack) package manager 15
 - null conditional operators 9, 299–305
 - dereferencing properties 299–300
 - handling Boolean comparisons 301–302
 - indexers and 302–303
 - limitations of 305
 - overview of 300–301
 - working with 303–304
 - event raising 303
 - null-returning APIs 304
 - null literals 44
 - null values 105
 - nullability checking 452–453
 - Nullable class 346
 - nullable integers 46
 - nullable intent 447–448
 - Nullable interface 44
 - nullable logic 46–48
 - nullable reference types 5, 440–453
 - advantages of 440–441
 - bang operator 445–447
 - changing meaning when using reference types 441–442
 - at compile time 443–445
 - entering 442–443
 - at execution time 443–445
 - future improvements to 449–453
 - deeper thinking about generics 451
 - enabling nullability checking 452–453
 - opt-in parameter validation 451–452
 - providing compilers with semantic information 450–451
 - migration of 447–449
 - best practices for using bang operators 448–449
 - end results 449
 - expressing nullable intent with attributes 447–448
 - iterations 448
 - null-inconsistent generics 449
 - Nullable struct 40–43
 - nullable value types 5, 38–49, 282
 - as operator and 48
 - expressing absence of information 39–40
 - language support 43–49
 - ? type suffix 43
 - conversions 44–45
 - lifted operators 45–46
 - null literal 44
 - nullable logic 46–48
 - null-coalescing ?? operator 48–49
 - Nullable struct 40–43
 - null-coalescing ?? operator 48–49, 452
 - null-inconsistent generics 449
 - NullPointerException 38
 - NullReferenceException 38, 43, 106, 299, 301, 305, 440, 443, 448
 - null-returning APIs 304
 - numeric literals, improvements to 429–431
 - binary integer literals 429–430
 - underscore separators 430–431

O

- object initializers 7, 81–86
 - benefits of single expressions for initialization 86
 - enhancements to 290–299
 - indexers in 291–294
 - overview of 81–83
 - test code vs. production code 298–299
- object-based collections 22
- object.Equals method 367

object-relational mapping (ORM) 119
ObsoleteAttribute 477
OnPropertyChanged method 226
op_Addition operator 228
OpenWrap project 15
OperationCanceledException 174, 179–180
 operations, retrying 311–312
 operators
 as operator 48
 conditional ?: operator 392–393
 default 34–37
 equality operators 337–338
 expression-bodied 245–247
 inequality 337–338
 is operator 372–374
 lifted 45–46
 null-coalescing ?? 48–49
 typeof 34–37
See also bang operators; null conditional operators
 opt-in parameter validation 451–452
 OR operator 46
 OrderBy method 108
 ordering
 comparisons
 regular equality and 342–343
 structural equality and 343–345
 evaluation order of pattern-based switch statements 377–379
 ORM. *See* object-relational mapping
 out parameters 427–428
 out variables 427–429
 inline variable declarations for out parameters 427–428
 restrictions lifted in C# 7.3 for 428–429
 out-of-process data 10
 overload resolution 436–437
 overloading 138
 with in parameters 398–399
 overview of 363–364

P

parallelism 189–190
 parameters 133–138
 in asynchronous methods 162
 with default values 134–135
 determining meaning of method calls 135–137
 impact on versioning 137–138
 adding overloads 138
 default value changes 137–138
 name changes 137
 opt-in parameter validation 451–452
 optional 140–142
 type parameters 26–28

See also ref parameters
 params modifier 135
 partial methods 68–69
 partial types 67–69
 pass-through properties 244–245
 pattern matching 353–380
 opportunities for 380
 overview of 365–367
See also recursive pattern matching
 pattern variable scopes 376–377
 pattern variables 428–429
 pattern-based fixed statements 413–414
 pattern-based switch statements 377–379
 patterns
 in C# 7.0 367–372
 constant 367–368
 of deconstruction 456–457
 with is operator 372–374
 matching properties in 455–456
 omitting types from 457–458
 with switch statements 374–379
 evaluation order of pattern-based switch statements 377–379
 guard clauses 375–376
 pattern variable scopes for case labels 376–377
 type 368–371
 var 371–372
 Paused state 195
 pausing 10, 195
 position, accessing elements by 325–326
 positional arguments 134–135
 pragma directives 72
 Preconditions.CheckNotNull 277
 predefined aliases 282
 primary constructors 468
 private protected 435, 477
 production code vs. test code 298–299
 projection initializer 87–88
 properties 235–251
 delegating 244–245
 dereferencing 299–300
 existing, deconstruction assignments to 357–361
 getter/setter access for 69–70
 history of 236–238
 matching in patterns 455–456
 pass-through 244–245
See also automatically implemented properties
 Properties property 294
 property declarations 8
 PropertyChangedBase class 226
 PropertyChangedEventArgs 225
 PropertyChangedEventHandler 224
 PropertyInfo 132, 282

Q

queries, LINQ 5
query expressions 107–110
 invocations 229–230
 LINQ syntax 110
 range variables 108–109
 translating from C# to C# 108
 transparent identifiers 108–109
Queryable class 111

R

raising events 303
range variables 108–109
ranges 458–460
 applying 459–460
 range literals 458–459
 range types 458–459
RanToCompletion 176
readability
 formatting for 274–275
 suggestions for 427
read-only automatically implemented properties 238–239
read-only computed properties 242–245
 delegating properties 244–245
 pass-through properties 244–245
 performing simple logic on another piece of state 245
read-only fields 419–420
read-only property 245
read-only structs 11
read-only variables
 implicit copying with 401–403
 references to 389–390
readonly
 declaring structs as 401–405
 modifier for structs 403–404
readonly modifier 403
ReadOnlyCollection 394
record types 6, 468–469
recursive pattern matching 455–458
 deconstruction patterns 456–457
 matching properties in patterns 455–456
 omitting types from patterns 457–458
redundant copying 11
reevaluating expressions 272–273
reexposing methods 297–298
ref extension methods 407–408
ref features 11
ref keyword 383, 385
ref locals 390–395
 conditional ?: operator in C# 7.2 392–393
 initializing 388–389

local variables 389
overview of 385
ref fields 389
ref readonly in C# 7.2 393–395
ref values in C# 7.2 392–393
references to read-only variables 389–390
 types 390
ref parameters 381–414
 in C# 7.2 405–408
 of enclosing methods 418
 overview of 382–385
ref readonly 393–395
ref returns 392–395
 conditional ?: operator in C# 7.2 392–393
 overview of 385–390
 ref readonly in C# 7.2 393–395
 ref values in C# 7.2 392–393
refactoring 88
reference conversion 145
reference types 22, 33, 74, 441–442
 See also nullable reference types
ReferenceEquals 450
reflection 132
ref-like structs, in C# 7.2 408–414
 IL representation of 414
 rules for 409–410
 Span 410–414
 stackalloc 410–414
regular equality 342–343
regular parameters 29
relational operators 45
release builds 194
resolution. *See* overload resolution
resource disposal 461–462
Result property 176
rethrowing exception filters 314–315
retry policies 311
retrying operations 311–312
return statement 167, 312
return types 161–162
return values, wrapping 166–168
returning 177
Roslyn code analyzer 348, 400
RuntimeBinderException 115–116, 123, 129, 131

S

safe awaiting 186
schemaless entity type 293
scopes
 capturing variables in 99–101, 422–423
 pattern variable 376–377
SecurityContext class 216
select clause 86
Select method 129

semantic information 450–451
 separators. *See* underscore separators
 serializing XML 404–405
SetException method 218
SetResult method 186, 218
SetStateMachine() method 199, 203–205, 210, 218
 short date 256
 signatures. *See* add signatures
 simple name 282
Sin method 288
 single expressions 86
 single value 326–328
 slow path 207
 source compatible 396
Span feature 11, 410–414
 specialized collections 22
SqlCommand constructor 266
SQLException 313
 stack trace 224
stackalloc 410–414, 477
 pattern-based fixed statements in C# 7.3 413–414
 with initializers in C# 7.3 413
StackOverflowException 208
Start() method 198, 200, 203
 state, performing logic on 245
 state machines 64, 195
 boxing 204–205
 structure of 199–202
 statement bodies 92
 statements. *See* fixed statements, pattern-based
 static classes 69
 static directives 284–290
 extension methods and 288–290
 importing static members 285–288
 static members 285–288
 static typing 77, 118–119
 string class 252–253
 string representations of tuples 341–342
stringArray 433
StringBuilder 291
StringCollection 22, 24–25
string.Format method 259
string.IsNullOrEmpty method 446, 450
string.IsNullOrWhiteSpace 470
 strings 252–283
 accessing identifiers with *nameof* 275–283
 common uses of *nameof* 277–279
 examples of 275–276
 tricks when using *nameof* 280–283
 formatting in .NET 253–258
 custom formatting with format strings 253–255
 localization 255–258
 simple string formatting 253

guidelines 270–275
 messages for end users 271–272
 messages for other developers 271
 handling 8–9
 interpolated string literals 258–261
 compiler handling of interpolated string literals 261
 formatting strings in interpolated string literals 259
 interpolated verbatim string literals 259–260
 simple interpolations 258–259
 limitations of 273–275
 deferring formatting for strings 273–274
 formatting for readability 274–275
 limitations of interpolated string literals 272–273
 localization using *FormattableString* 261–270
 compiler handling of interpolated string literals 262–263
 formatting *FormattableString* in specific cultures 263–265
 uses for *FormattableString* 265–268
 using *FormattableString* with older versions of .NET 268–270
 machine-readable 270–271
 strongly typed 77
 structs
 automatically implemented properties in 240–242
 declaring as *readonly* in C# 7.2 401–405
 implicit copying with read-only variables 401–403
 XML serialization implicitly read-write 404–405
 nongeneric *ValueTuple* struct 346
readonly modifier for 403–404
See also ref-like structs, in C# 7.2
 structural equality 343–345
 stub method 195, 198–199
Substring method 126, 459
 switch expressions 453–455
 switch statements 197, 366
 pattern-based 377–379
 patterns with 374–379
 guard clauses 375–376
 pattern variable scopes for case labels 376–377
 synchronization contexts 157–158
SynchronizationContext class 157
 synchronous code 190
SynonymInfo indexer 142
System.Collections.CollectionBase class 25
System.Collections.Generic 282
System.ComponentModel namespace 224
System.Diagnostics.StackTrace 224
System.Linq namespace 230

System.Linq.Enumerable class 107, 289
System.Linq.Expressions namespace 102
System.Linq.Queryable class 289
System.Object.ToString() method 259
System.Runtime.CompilerServices
 namespace 127, 223, 232, 340, 396, 414
System.Runtime.CompilerServices.AsyncMethodBuilderAttribute 184
System.Runtime.CompilerServices.FormattableStringFactory class 263
System.Runtime.CompilerServices.Unsafe
 package 404
System.Runtime.INotifyCompletion interface 163
System.Runtime.InteropServices namespace 395
System.Threading.Tasks.Extensions package 182
System.Tuple 347
System.TupleExtensions class 346
System.ValueTuple 338–339

T

target-typed new 470
Task class 155
Task Parallel Library (TPL) 151, 179
Task property 218
task types 161
 in C# 7 182–186
 building 184–186
 ValueTask 182–184
 custom 218–219
TaskAwaiter 217
task-based asynchronous pattern 151
TaskCanceledException 174
Task.ConfigureAwait 170
Task.Delay 170, 212
Task.FromCanceled 191
Task.FromException 191
Task.FromResult 170, 191, 202
Task.Result 158
Task.Result property 190
Task.Wait() method 158, 190
Task.WhenAll() method 175
Task.Yield() method 164
test code vs. production code 298–299
TestCaseSource attribute 279–280
testing asynchrony 191–192
this keyword 103, 475
ThreadAbortException 208
throw expressions 431–432
throw statement 312, 314
ThrowIfCancellationRequested 179
Title property 119
tokens 156
ToList() method 148
ToString() method 33, 68, 89, 249, 341–342

ToString(IFormatProvider) method 263, 265
ToString(string, IFormatProvider) method 32
TPL. *See* Task Parallel Library
trampoline technique 215
translating query expressions 108
transparent identifiers 108–109
true operator 45
Try .NET 14
try block 59
try statement 205
TryAdd method 293
try/catch block 208
try/finally blocks 213–216
TryGetNext method 463, 465
Tuple class 28, 31
tuple expression 355
tuple literals 321–328
 conversions 332–334
 conversions to tuples types from 330–334
 explicit conversions 331–332
 implicit conversions 330–331
deconstruction 361
inferred element names for in C# 7.1 323–324
syntax 321–323
types of 329–330
tuple types 321–338
 conversions between 334–336
 generic variance conversions 336
 tuple type identity conversions 335–336
 conversions from tuple literals 330–334
 element name checking in inheritance 336–337
 equality operators in C# 7.3 337–338
 inequality operators in C# 7.3 337–338
 syntax 321–323
Tuple<,> class 320
TupleElementNamesAttribute 340
tuples 6, 319–352
 alternatives to 346–348
 anonymous types 347–348
 named types 348
 System.Tuple 347
 as bags of variables 324–328
 accessing elements by name 325–326
 accessing elements by position 325–326
 in CLR 338–346
 element name handling 339–341
 extension methods 346
 nongeneric ValueTuple struct 346
 regular equality and ordering
 comparisons 342–343
 string representations of tuples 341–342
 structural equality and ordering
 comparisons 343–345
 System.ValueTuple 338–339
wombles 345

tuples (*continued*)

- conversions 329–338
 - element name checking in inheritance 336–337
 - equality operators in C# 7.3 337–338
 - inequality operators in C# 7.3 337–338
 - uses of 336
- deconstruction of 354–361
- dynamic binder and 351–352
 - element names 351–352
 - high element numbers 352
- fields 350–351
- implementing conversions 341
- large 345
- local variables 349–350
- nonpublic APIs 348–349
 - overview of 320
 - as single value 326–328
- two-pass exception model 307–310
- type arguments
 - to methods 30–32
 - overview of 26–28
- type classes 35, 469
- type constraints
 - generic 435–436
 - overview of 32–34
- type inference 30–32
- Type Library Importer tool (tlbimp) 139
- type parameters 26–28
- type patterns 368–371
- typeof operator 34–37, 275, 409
- types 4–6, 390
 - arrays 282
 - compiler-generated 89–90
 - exceptions 310
 - indexes 458–459
 - named 348
 - omitting from patterns 457–458
 - range 458–459
 - record 468–469
 - reference 441–442
 - referring to members of 280
 - return 161–162
 - static 118–119
- See also* anonymous types; generic types; nontuple types, deconstruction of; nullable reference types; nullable value types; partial types; task types; tuple types
- type-to-type conversion 334
- typing
 - explicit 77
 - static 77
 - terminology of 77
- See also* dynamic typing; implicit typing

U

-
- unary operator 46, 459
 - underlying type 41
 - underscore separators 430–431
 - unit-test frameworks 191
 - Unity 14
 - Universal Windows Applications (UWA/UWP) 151
 - unmanaged constraint 478
 - UnmanagedType enum 478
 - unnamed arguments 434
 - unsafe awaiting 186
 - unspeakable names 17, 237
 - unwrapping exceptions 174–176
 - users. *See* end users
 - using await statement 461–462
 - using statement 205, 310
 - using static directive 286, 288, 314, 475
 - UWA/UWP. *See* Universal Windows Applications

V

-
- validation
 - arguments 177–179, 277
 - iterator/async arguments 426–427
 - See also* opt-in parameter validation
 - Value property 42
 - value types 33, 198
 - See also* nullable value types
 - values
 - ref values in C# 7.2 392–393
 - tuples as single values 326–328
 - wrapping return values 166–168
 - See also* default values; dynamic values
 - ValueTask 182–184
 - ValueTuple constructor 341
 - ValueTuple struct, nongeneric 346
 - ValueTuple types 476
 - var keyword 78–79, 87
 - var pattern 371–372
 - variable declarations 427–428
 - variables
 - captured 417–418
 - capturing 94–101
 - from multiple scopes 99–101
 - implementing with generated class 95–97
 - in foreach loops 220–222
 - in scopes 417
 - existing, deconstruction assignments to 357–361
 - within local methods 417–420
 - capturing ref parameters of enclosing methods 418
 - declaring local method after declaring captured variables 417–418

- local methods and read-only fields 419–420
local methods interacting with definite assignment 419
new, deconstruction to 355–357
range 108–109
tuples as bags of 324–328
 accessing elements by name 325–326
 accessing elements by position 325–326
See also local variables; out variables
- variance
 conversions 336
 in delegate declarations 144–145
 in interface declarations 144–145
 restrictions on using 145–147
See also generic variance
- VARIANT type 140
- versioning, impact of parameters on 137–138
 adding overloads 138
 default value changes 137–138
 parameter name changes 137
- void method 167, 177, 246, 362
- WebAssembly 14
WebClient 154, 156
WebClient.DownloadStringAsync method 154
WebException 310, 314
WebRequest 24
WhenAll() method 175
where clause 33
Where method 301
Windows Forms 13
Windows Presentation Foundation (WPF) 13
Windows Runtime platform (WinRT) 151
wombles 345
wrapping return values 166–168

X

- Xamarin Forms 14
 XElement 44, 450
 XML serialization 404–405
 XNamespace 116

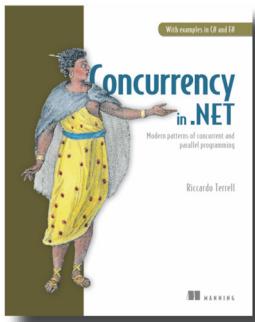
Y

- yield statements 54, 56–57, 59, 160, 465
yield type 54
Yield() method 164
YieldAwaitable 164

W

- Wait() method 157, 174, 179
WaitForNextAsync method 463, 465
weakly typed 77

RELATED MANNING TITLES



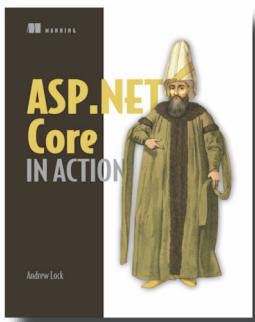
Concurrency in .NET

Modern patterns of concurrent and parallel programming
by Riccardo Terrell

ISBN: 9781617292996

568 pages, \$59.99

June 2018



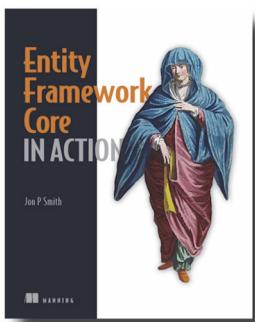
ASP.NET Core in Action

by Andrew Lock

ISBN: 9781617294617

712 pages, \$49.99

June 2018



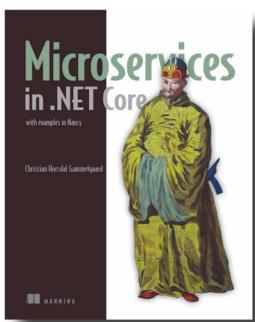
Entity Framework Core in Action

by Jon P Smith

ISBN: 9781617294563

520 pages, \$49.99

July 2018



Microservices in .NET Core

with examples in Nancy

by Christian Horsdal Gammelgaard

ISBN: 9781617293375

344 pages, \$49.99

January 2017

For ordering information go to www.manning.com

C# IN DEPTH Fourth Edition

Jon Skeet

The powerful, flexible C# programming language is the foundation of .NET development. Even after two decades of success, it's still getting better! Exciting new features in C# 6 and 7 make it easier than ever to take on big data applications, cloud-centric web development, and cross-platform software using .NET Core. There's never been a better time to learn C# in depth.

C# in Depth, Fourth Edition is a revised edition of the bestseller written by C# legend Jon Skeet. This authoritative and engaging guide is your key to unlocking this powerful language, including the new features of C# 6 and 7. In it, Jon introduces expression-bodied members, interpolated strings, pattern matching, and more. Real-world examples drive it all home. By the end of this awesome book, you'll be writing C# code with skill, style, and confidence.

What's Inside

- Comprehensive coverage of C# 6 and 7
- Greatest hits of C# 2–5
- Extended pass-by-reference functionality
- String interpolation
- Composition with tuples
- Decomposition and pattern matching

For intermediate C# developers.

Jon Skeet is a senior software engineer at Google. He studied mathematics and computer science at Cambridge, is a recognized authority in Java and C#, and maintains the position of top contributor to Stack Overflow.

To download their free eBook in PDF, ePUB, and Kindle formats, owners of this book should visit
www.letmerefined.net



See first page

“Jon doesn't just explain how C# works; he explains how the whole thing holds together as a unified design, and also points out when it doesn't.”

—From the Foreword by Eric Lippert, Facebook

“Provides an excellent overview of the evolution of C# with helpful and realistic examples that make learning the newest features of C# easy.”

—Meredith Godar, Innovative Software Engineering

“This book has it all—from the beginnings of C# to insights on the future of the language and everything in between!”

—Willem van Ketwich
 National Australia Bank

ISBN-13: 978-1-61729-453-2
 ISBN-10: 1-61729-453-5

5 4 9 9 9



9 781617 294532



MANNING

\$49.99 / Can \$65.99 [INCLUDING eBOOK]